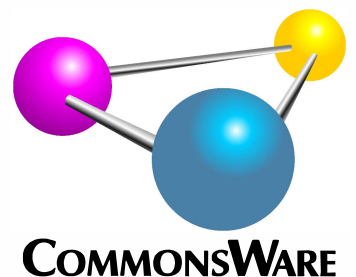


Version 2.2

Elements of Android Jetpack

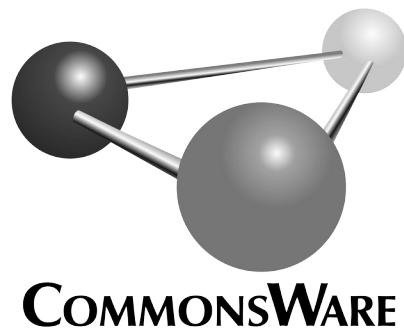


Mark L. Murphy



Elements of Android Jetpack

by Mark L. Murphy



Elements of Android Jetpack
by Mark L. Murphy

Copyright © 2019-2021 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

Printing History:
October 2021: Version 2.2

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	
◦ Prerequisites	xi
◦ About the Updates	xi
◦ <i>What's New in Version 2.2?</i>	<i>xii</i>
◦ Warescription	xii
◦ Source Code and Its License	xiii
◦ <i>Creative Commons and the Four-to-Free (42F) Guarantee</i>	<i>xiii</i>
◦ Acknowledgments	xiv
• Introducing Android	
◦ Focus: Apps, Not Operating Systems	1
◦ <i>What You Need</i>	<i>1</i>
◦ How This Book Is Organized	6
• Setting Up the Tools	
◦ <i>But First, Some Notes About Android's Emulator</i>	<i>7</i>
◦ <i>Step #1: Install Android Studio</i>	<i>9</i>
◦ <i>Step #2: Running Android Studio for the First Time</i>	<i>10</i>
• Getting Your First Project	
◦ <i>Step #1: Importing the Project</i>	<i>19</i>
◦ <i>Step #2: Get Ready for the x86 Emulator</i>	<i>21</i>
◦ Step #3: Set Up the AVD	22
◦ Step #4: Set Up the Device	26
◦ Step #5: Running the Project	32
• Taking a Tour of Android Studio	
◦ The Project Tree	36
◦ The Editing Pane	39
◦ The Docked Views	39
◦ Popular Menu and Toolbar Options	40
◦ Android Studio and Release Channels	47
• Examining Your Code	
◦ The Top Level	51
◦ <i>The Project Contents</i>	<i>52</i>
◦ The App Module Contents	54
◦ The Generated Source Sets	55
◦ <i>Language Differences</i>	<i>59</i>
◦ Introducing the Activity	59

◦ Other Things in the Project Tree	65
• Exploring Your Resources	
◦ What You See in res/	68
◦ OS Versions and API Levels	68
◦ Decoding Resource Directory Names	71
◦ Our Initial Resource Types	71
◦ About That R Thingy	77
◦ The Resource Manager	78
• Inspecting Your Manifest	
◦ The Root Element	81
◦ The Application Element	82
◦ The Activity Element (And Its Children)	83
• Reviewing Your Gradle Scripts	
◦ Gradle: The Big Questions	85
◦ Obtaining Gradle	87
◦ Examining the Gradle Files	89
◦ Requesting Plugins	93
◦ Android Plugin for Gradle Configuration	94
◦ Other Stuff in the android Closure	96
◦ Libraries and Dependencies	97
• Inspecting the Compiled App	
◦ What We Build	99
◦ Where They Go	101
◦ Building the APK	101
◦ Analyzing the APK	102
• Touring the Tests	
◦ Instrumented Tests	105
◦ Unit Tests	113
• Introducing Jetpack	
◦ What, Exactly, is Jetpack?	117
◦ Um, OK, So, What's the Point?	117
◦ Key Elements of Jetpack	118
◦ What Came Before: the Android Support Library	122
• Introducing the Sampler Projects	
◦ The Projects	125
◦ Getting a Sampler Project	126
◦ The Modules	129
◦ Running the Samples	129
• Starting Simple: TextView and Button	
◦ First, Some Terminology	133
◦ Introducing the Graphical Layout Editor	138

◦	TextView: Assigning Labels	146
◦	Button: Reacting to Input	152
◦	The Curious Case of the Missing R	161
•	Debugging Your App	
◦	Get Thee To a Stack Trace	164
◦	Running Your App in the Debugger	167
◦	So, Where Did We Go Wrong?	173
•	Introducing ConstraintLayout	
◦	The Role of Containers	178
◦	Layouts and Adapter-Based Containers	179
◦	ConstraintLayout: One Layout To Rule Them All	180
◦	Getting ConstraintLayout	181
◦	Using Widgets and Containers from Libraries	181
◦	A Quick RTL Refresher	183
◦	Simple Rows with ConstraintLayout	184
◦	Starting from Scratch	188
◦	ConstraintLayout and the Attributes Pane	190
◦	EditText: Making Users Type Stuff	191
◦	More Complex Forms	193
◦	Turning Back to RTL	200
◦	More Fun with ConstraintLayout	201
◦	Notes on the Classic Containers	202
•	Integrating Common Form Widgets	
◦	ImageView and ImageButton	205
◦	Compound Buttons	211
◦	SeekBar	223
◦	ScrollView: Making It All Fit	226
◦	Other Notes About the Sample	229
•	Contemplating Contexts	
◦	It's Not an OMG Object, But It's Close	237
◦	The Major Types of Context	238
◦	Key Context Features	240
◦	Know Your Context	242
◦	Context Anti-Patterns	243
•	Icons	
◦	App Icons... And Everything Else	245
◦	Creating an App Icon with the Asset Studio	246
◦	Creating Other Icons with the Asset Studio	252
•	Adding Libraries	
◦	Depending on a Local JAR	253
◦	Artifacts and Repositories	254

◦ <i>Requesting Dependencies</i>	255
• Employing RecyclerView	
◦ Recap: Layouts vs. Adapter-Based Containers	259
◦ The Challenge: Memory	260
◦ Enter RecyclerView	261
◦ <i>A Trivial List</i>	262
◦ Hey, What About ListView?	279
◦ Gesture Navigation and Scrolling Widgets	280
• Coping with Configurations	
◦ What's a Configuration? And How Do They Change?	283
◦ Configurations and Resource Sets	284
◦ Implementing Resource Sets	285
◦ Resource Set Rules	288
◦ Activity Lifecycles	292
◦ When Activities Die	296
◦ Context Anti-Pattern: Outliving It	297
• Integrating ViewModel	
◦ Configuration Changes	299
◦ What We Want... and What We Do Not Want	300
◦ Enter the ViewModel	301
◦ <i>Applying ViewModel</i>	301
◦ ViewModel and the Lifecycle	306
◦ Changing Data in the ViewModel	310
◦ ViewModel and AndroidViewModel	322
◦ ViewModelFactory	323
• Understanding Processes	
◦ When Processes Are Created	325
◦ What Is In Your Process	326
◦ BACK, HOME, and Your Process	326
◦ Termination	327
◦ Foreground Means "I Love You"	328
◦ <i>Tasks and Your App</i>	329
◦ Instance State	331
◦ Pondering Parcelable	333
◦ A State-Aware ViewModel	337
• Binding Your Data	
◦ <i>The Basic Steps</i>	347
◦ Why Bother?	357
◦ The Other Common "Gimme the Views" Options	358
• Defining and Using Styles	
◦ Styles: DIY DRY	361

◦ Elements of Style	362
◦ Themes: Would a Style By Any Other Name... ..	366
◦ Android 10 Dark Mode	371
◦ The DayNight Solution	375
◦ The Material Components for Android	376
◦ Context Anti-Pattern: Using Application Everywhere	377
• Configuring the App Bar	
◦ So. Many. Bars.	380
◦ Vector Drawables	382
◦ Menu Resources	387
◦ Using Toolbar Directly	392
◦ Using Toolbar as the Action Bar	405
◦ Having Fun at Bars	408
• Implementing Multiple Activities	
◦ Multiple Activities, and Your App	409
◦ Creating Your Second (and Third and...) Activity	410
◦ Starting Your Own Activity	412
◦ Extra! Extra!	413
◦ Seeing This In Action	413
◦ Using Implicit Intents	418
◦ Activities and Results	420
◦ The Inverse: <intent-filter>	429
• Adding Library Modules	
◦ Reasons for Library Modules	431
◦ Consuming a Library Module	433
• Adopting Fragments	
◦ The Six Questions	435
◦ Where You Get Your Fragments From	440
◦ Static vs. Dynamic Fragments	440
◦ Fragments, and What You Have Seen Already	441
◦ ToDo, or Not ToDo? That Is the Question	441
◦ The Fragment Lifecycle Methods	469
◦ View Binding and Fragments	471
◦ Context Anti-Pattern: Assuming Certain Types	477
• Navigating Your App	
◦ What We Get from the Navigation Component	479
◦ Elements of Navigation	483
◦ A Navigation-ized To-Do List	490
◦ So... Was It Worth It?	502
◦ The Next Wave: Kotlin DSL	502
• Dialogs	

◦ A Tale of Four Dialogs	509
◦ <i>Using AlertDialog and DialogFragment</i>	512
• Writing an App Widget	
◦ Writing a... What?	519
◦ Challenges with App Widgets	521
◦ Introducing Broadcasts and Receivers	523
◦ Pieces of the App Widget	523
◦ Colors in an App Widget	525
◦ Actions and App Widgets	533
• Thinking About Threads and LiveData	
◦ The Main Application Thread	541
◦ The UI Thread is for UI	543
◦ Introducing LiveData	544
◦ Colors... Live!	546
◦ Sources of Owners	552
◦ Where Do Threads Come From? Um, Besides From Me?	553
◦ Coroutines and ViewModel	555
◦ <i>The End of LiveData?!?</i>	556
• Adding Some Architecture	
◦ Repositories	557
◦ Unidirectional Data Flow	558
◦ <i>A UDF Implementation</i>	560
◦ <i>The Kotlin LiveData Alternative: StateFlow</i>	581
◦ <i>States and Events</i>	584
• Working with Content	
◦ The Storage Access Framework	591
◦ Android 11+ Restrictions	603
• Using Preferences	
◦ The Preferred Preferences	605
◦ Collecting Preferences with PreferenceFragmentCompat	606
◦ Types of Preferences	612
◦ Working with SharedPreferences	615
• Requesting Permissions	
◦ Frequently-Asked Questions About Permissions	621
◦ Dangerous Permissions: Request at Runtime	624
• Handling Files	
◦ The Three Types of File Storage	631
◦ What the User Sees	634
◦ Storage, Permissions, and Access	634
◦ <i>Reading, Writing, and Debugging Storage</i>	635
◦ Serving Files with FileProvider	654

◦ What You Should Use	666
• Accessing the Internet	
◦ An API Roundup	667
◦ Android's Restrictions	670
◦ Forecasting the Weather	673
• Inverting Your Dependencies	
◦ The Problem: Test Control	693
◦ The Solution: Dependency Inversion	694
◦ Dependency Inversion in Android	695
◦ Applying Koin	695
• Storing Data in a Room	
◦ Room Requirements	708
◦ Room Furnishings	708
◦ Other Fun Stuff in the App	715
◦ What Else Does Room Offer?	725
◦ Examining Your Database	726
• Testing Your Changes	
◦ A Quick Recap	731
◦ Which Tests Should I Write?	732
◦ Writing Unit Tests	732
◦ Employing Mocks	737
◦ Writing Instrumented Tests	744
◦ Writing Basic Espresso Tests	749
◦ Another Option: UI Automator	754
◦ Again: What Should I Be Using?	755
• Working with WorkManager	
◦ The Role of WorkManager	757
◦ WorkManager Dependencies	758
◦ Workers: They Do Work	759
◦ Performing Simple Work	762
◦ Work Inputs	763
◦ Constrained Work	764
◦ Tagged Work	765
◦ Monitoring Work	766
◦ Canceling Work	774
◦ Delayed Work	775
◦ Parallel Work	775
◦ Chained Work	776
◦ Periodic Work	786
◦ Unique Work	786
◦ Testing Work	787

◦ Inspecting Work	791
◦ WorkManager and Side Effects	796
• Creating a New Project	
◦ Key Decisions That You Need to Make	801
◦ The New-Project Wizard	804
◦ Copying an Existing Project	808
◦ Creating a Library Module	809
• Signing Your App	
◦ Role of Code Signing	813
◦ What Happens In Debug Mode	814
◦ Production Signing Keys	815
• Shrinking Your App	
◦ Why We Care	823
◦ Identify What to Attack	824
◦ Shrinking Your Dependencies	824
◦ Shrinking Your Code	826
◦ Removing Unused Resources	830
◦ Optimizing Bitmaps	832
◦ Hey, What About App Bundles?	837
• Using the AVD Manager and the Emulator	
◦ Notable AVD Configuration Options	839
◦ The Emulator Sidebar	845
◦ Emulator Window Operations	861
◦ In-IDE Emulator	862
• Using the SDK Manager	
◦ Installing Platform Pieces	868
◦ Installing and Upgrading Tools	870
◦ Adding Third-Party SDK Suppliers	871
• Configuring Your Project	
◦ Risks and Rewards	873
◦ The Project Category	874
◦ The SDK Location	875
◦ The Variables	876
◦ The Modules	877
◦ Dependencies	880
◦ Build Variants	881
◦ Suggestions	883
• Configuring Android Studio	
◦ Searching for Settings	886
◦ Themes and Colors	887
◦ Fonts. And Other Fonts.	889

◦ Code Styles	893
◦ Inlay Hints	898
◦ Other Settings of Note	900
• Coping with New Android Versions	
◦ The March of the Versions	903
◦ The Typical Release Process	904
◦ Things to Worry About	906
• Deciding Where to Go From Here	
◦ The Rest of the Books	909
◦ Android Developer Support	910
◦ Major Conferences	910

Preface

Thanks!

Thanks for your interest in developing applications for Android! Android has grown from nothing to 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.

Prerequisites

This book is written for developers with prior experience in Java or Kotlin. If you are not familiar with either of these languages, it will be difficult for you to follow the code samples in the book. The author of this book has published [Elements of Kotlin](#). If you obtained this book via [the Warescription](#), you are eligible to download a copy of *Elements of Kotlin*, along with the rest of the CommonsWare line of books.

This book is written for people who have used an Android device before. If you are not familiar with basic Android concepts — such as the home screen and launcher, navigating home and back, and so on — you will want to spend time with an Android device.

About the Updates

This book will be updated a few times per year, to reflect newer versions of Android,

Android Studio, and the Jetpack family of libraries.

If you obtained this book through [the Warescription](#), you will be able to download updates as they become available, for the duration of your subscription period.

If you obtained this book through other channels... um, well, it's still a really nice book!

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

- The Table of Contents in the ebook formats (PDF, EPUB, MOBI/Kindle) shows sections with changes in ***bold-italic*** font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

And, there is the “What’s New” section, just below this paragraph.

What’s New in Version 2.2?

This is a minor update, primarily to support Android Studio 2020.3.1 Arctic Fox. |

This update adds coverage of using StateFlow as an alternative to using LiveData in Kotlin-based Android projects. It also replaces BroadcastChannel with SharedFlow in the one Kotlin sample that used SharedFlow. |

And, in addition to all of that, there are the usual roster of bug fixes, to the prose and to the sample code. |

Warescription

If you purchased the Warescription, read on! If you obtained this book from other channels, feel free to [jump ahead](#).

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, plus the ability to read the book online at [the Warescription Web site](#). You also have access to other books that CommonsWare publishes during that subscription period.

PREFACE

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.

However, you can only download the books while you have an active Warescription. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The [CommonsBlog](#)
2. The [CommonsWare](#) Twitter feed
3. Opting into emails announcing each book release — log into the [Warescription](#) site and choose Configure from the nav bar
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.

Source Code and Its License

The source code in this book is licensed under the [Apache 2.0 License](#), in case you have the desire to reuse any of it.

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 version will be available for use under the [Creative Commons Attribution-Noncommercial-ShareAlike 3.0](#) license as of the fourth

PREFACE

anniversary of its publication date. That means that, once four years have elapsed, you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community.

This edition of this book will be available under the aforementioned Creative Commons license on *1 October 2025*.

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. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

The author would like to thank the Google team responsible for Android and the Android Jetpack.

Part One: Getting Started

Introducing Android

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.

Focus: Apps, Not Operating Systems

This book is focused on writing Android applications (“apps”). An app is something that a user might install from the Play Store or otherwise download to their device. That app usually has some user interface, and it might have other code designed to work in the background.

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](#).

What You Need

In order to get started as an Android developer — and to get the most out of this book — you are going to need several things, outlined in the following sections.

A Development Machine

For the purposes of this book, you will need a Windows, macOS, or Linux computer on which to write your Android apps. This is how the vast majority of Android app developers do their work, though there are tools (e.g., AIDE) that allow you to write

Android apps directly on an Android device.

Your development machine should be as powerful as you can manage:

- A fast CPU (e.g., quad-core Intel Core i5/i7/i9 with at least 2.0 GHz clock speed per core)
- As much RAM as you can manage (8GB minimum, 16GB or more recommended)
- As fast of a hard drive as you can find (an SSD is an excellent choice in general)
- A screen with enough resolution to use the development tools (1280x800 minimum resolution)

The primary development tool for Android apps — called Android Studio — consumes a lot of resources, particularly when compiling a project, which is why it helps to have a powerful development machine.

Note that Apple M1 chip support is still a work in progress. As of September 2021, Android Studio Arctic Fox has preliminary support for the M1 chip. This should improve with time.

Language Experience

In general, to write Android apps, you need to know how to work with computer programming languages. In particular, Android app development is focused heavily on Java and Kotlin, with Groovy also playing a role.

Java

The original programming language used for Android app development was Java. Right now, most Android code in the world is written in Java, and most educational material is written around Java.

As a result, to be an Android app developer today, it helps to know Java.

This book does not teach you Java. Java has been around for around two decades, and so there are *lots* of existing books, courses, videos, and the like to help you learn Java. However, there are many things in Java that are not really relevant for Android app development, such as Swing desktop GUIs and Java servlets for Web applications. 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.

Kotlin

The primary current language for Android app development is Kotlin. Kotlin is a fairly new language, having only reached 1.0 status in 2016. That causes some problems, as there is less material about how to write Kotlin than there is on how to write Java. On the other hand, Kotlin adopts newer approaches and discards legacy “cruft”. The resulting language can be much more concise, getting more work done with fewer lines of code.

This book does not teach you Kotlin. The author of this book is also the author of [Elements of Kotlin](#), which was written with an eye towards it being a companion to the book that you are reading now. From time to time, you will find this book pointing out relevant chapters and sections in *Elements of Kotlin*, to help newcomers to both Android *and* Kotlin learn both subjects.

Note that at the 2019 Google I/O conference, Google indicated that the Android SDK will be “Kotlin first” going forward. While Java development is still possible, Google will be focusing on Kotlin in terms of documentation, samples, education, and some new technologies. So, while this book will present material in both Java and Kotlin, you should strongly consider learning Kotlin in the not-too-distant future.

Groovy and Gradle

The code that causes your app to do stuff will be written mostly in Java and Kotlin.

The code that causes your app to be *built* out of that Java and Kotlin will be written in Groovy... though you may not notice this much.

Most Android apps are built using a build tool called Gradle. Gradle is a program for building other programs. We will be working with “Gradle build scripts” to configure how Gradle turns our source code into a (hopefully) working app. The Gradle build scripts that we use today usually are written using the Groovy programming language.

However, for most basic uses of Gradle — including pretty much everything in this book — you will not need to think much about Groovy syntax. Just follow the recipes described in the book, and you can put off learning Groovy until such time as you really want to start creating elaborate build scripts.

An Android Test Environment

Writing Android apps is fun!

(no, really!)

However, that fun only appears when you can actually run the app that you created. Otherwise, you just have a hunk of source code that sits around doing nothing. To run the app, you will need an Android device or emulator.

Devices

Every Android developer should have at least one Android device. Every Android device that legitimately has the Play Store on it is able to be used for app development. You can enable the super-secret “developer options” in the device, to allow you to install apps that you have written yourself on the device and test them out — we will see how to do that in this book.

Typically, that Android device will be a phone, though you could test on something else, such as a tablet, if you wish. Android app development puts few requirements on the device itself; for example, you do not necessarily need to have a usable SIM installed in the phone.

In [an upcoming chapter](#), you will see how to configure your Android device for use with app development.

Emulators

All Android developers should have at least one device. Some Android developers, such as the author of this book, have lots of Android devices. However, inevitably, you run into cases where hardware is a problem:

- You want to test your app on different versions of Android, but you do not have a device for a particular Android version
- You want to test your app for various screen sizes and resolutions, but you do not have devices for all of the scenarios that you wish to test
- You want to test your app in unusual situations, such as running on a Chromebook, and you do not have a device that matches

For those cases, the Android tools come with an emulator. The emulator gives you an app for your development machine that pretends to be an Android device. You decide what sort of device it is: Android version, screen size and resolution, and so on. You can run your app on the emulator and get a sense for what it would be like for the app to be running on a real device with those same characteristics.

In [an upcoming chapter](#), you will see how to set up the Android emulator.

Patience and Serenity

Android app development often can be a frustrating experience:

- Advice that you get from older sources may not work, due to changes in Android
- Dealing with multiple programming languages makes it more difficult to make use of advice that you get, if you have to keep converting code snippets between languages
- The GUI that you wrote that works fine on one device does not work quite as well on the next device
- And so on

You will be able to address all of these challenges in time. Early on, though, you should expect that these sorts of problems will arise, and you will need to cope with them when they do.

How This Book Is Organized

This book is divided into two major parts: a “Hello, World” walk-through and “deeper dives” into major Android app development topics.

“Hello, World!”, Front to Back

For decades, the classic first program for a person in a given programming environment is dubbed [“hello, world”](#).

In a typical programming environment, a “hello, world!” app is usually trivial, offering little to learn from.

Android is complicated, which makes even “hello, world!” a place to learn all sorts of things, from how user interfaces get constructed, to how our tools work, to how the tools know how to build that app and show us that user interface.

So, in the first several chapters, we will examine various facets of a “hello, world!” app generated from Android Studio’s new-project wizard.

Deeper Dives

Of course, a “hello, world!” app is very shallow. You will not get very far in Android app development if all you know is what “hello, world!” shows you.

So, after the tour of the “hello, world!” app, we will expand upon the concepts seen there, exploring different aspects of Android app development. We will not cover *everything* in Android — that would take thousands upon thousands of pages. This book will give you a basic foundation for Android app development and will help point you to places to learn other facets of what Android apps can do.

Setting Up 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

As mentioned in the previous chapter, 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.

Emulators not only emulate Android itself, but also the CPU of the Android device. Most Android devices have ARM CPUs... but it is likely that your development machine has an x86 CPU. The emulator can emulate an ARM CPU when running on your x86 CPU, but it is slow. Fortunately, the emulator can also emulate an Android device that has an x86 CPU, and this runs much more quickly. You *really* want to be able to use the x86 emulator.

However, to use the x86 emulator, you will need some additional software:

- Linux users need KVM
- macOS and Windows users need the “Intel Hardware Accelerated Execution Manager” (a.k.a., HAXM), which the Android Studio installer will attempt to install for you

SETTING UP THE TOOLS

And, the x86 emulator will only work for CPUs meeting certain requirements:

Development OS	CPU Manufacturer	CPU Requirements
macOS	Intel	any modern Mac should work
macOS	Apple M1	unclear
Linux/ Windows	Intel	support for Intel VT-x, Intel EM64T (Intel 64), and Execute Disable (XD) Bit functionality
Linux	AMD	support for AMD Virtualization (AMD-V) and Supplemental Streaming SIMD Extensions 3 (SSSE3)
Windows 10 April 2018 or newer	AMD	support for Windows Hypervisor Platform (WHPX) functionality

If your CPU does not meet those requirements, you will want to have one or more Android devices available to you, so that you can test on hardware rather than the emulator.

If you are running Windows or Linux, you need to ensure that your computer's BIOS is set up to support "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". Enable them if they are not already enabled. macOS machines come with virtualization extensions pre-enabled.

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.

Note that Apple M1 chip support is still a work in progress. While Android Studio Arctic Fox appears to have M1 support, it also appears that this support is a bit rough in spots. Hopefully, this will smooth out with future versions of Android

Studio.

Step #1: Install Android Studio

At the time of this writing, the current production version of Android Studio is 2020.3.1 Arctic Fox and this book covers that version. Android Studio gets updated often, and so you may be on a newer version — 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](#). Or, you can download Android Studio Arctic Fox directly, for:

- [Windows](#)
- [macOS x86](#)
- [macOS M1](#)
- [Linux](#)

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

Mac x86 users can download a DMG disk image and install it akin to other Mac software, dragging the Android Studio icon into the Applications folder. M1 users get a ZIP file instead.

Linux 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 in your Android Studio installation's `bin/` directory.

Step #2: Running Android Studio for the First Time

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

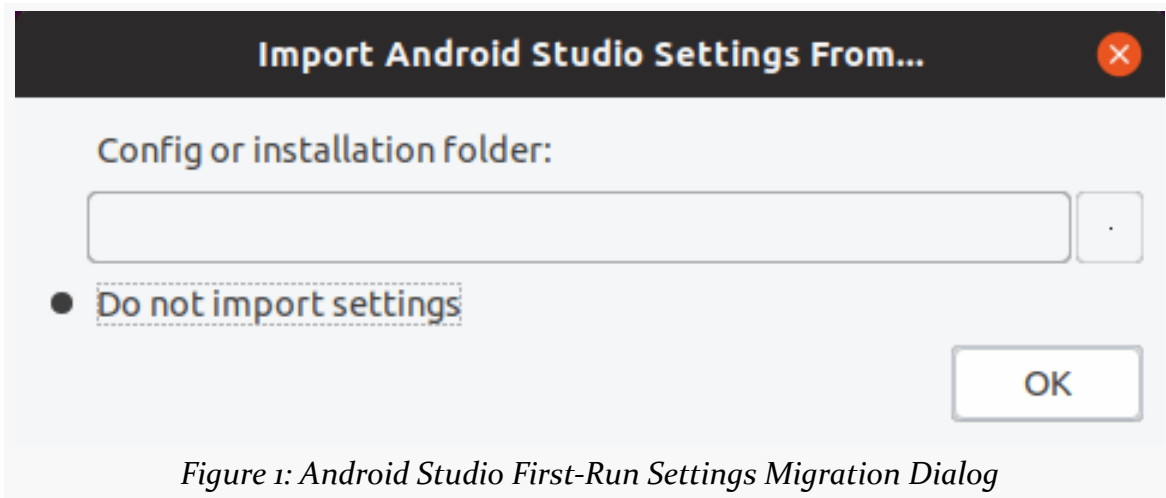


Figure 1: Android Studio First-Run Settings Migration Dialog

If you are using Android Studio for the first time, the “Do not import settings” option is the correct choice to make.

SETTING UP THE TOOLS

Then, after a short splash screen, you may be presented with a “Data Sharing” dialog:

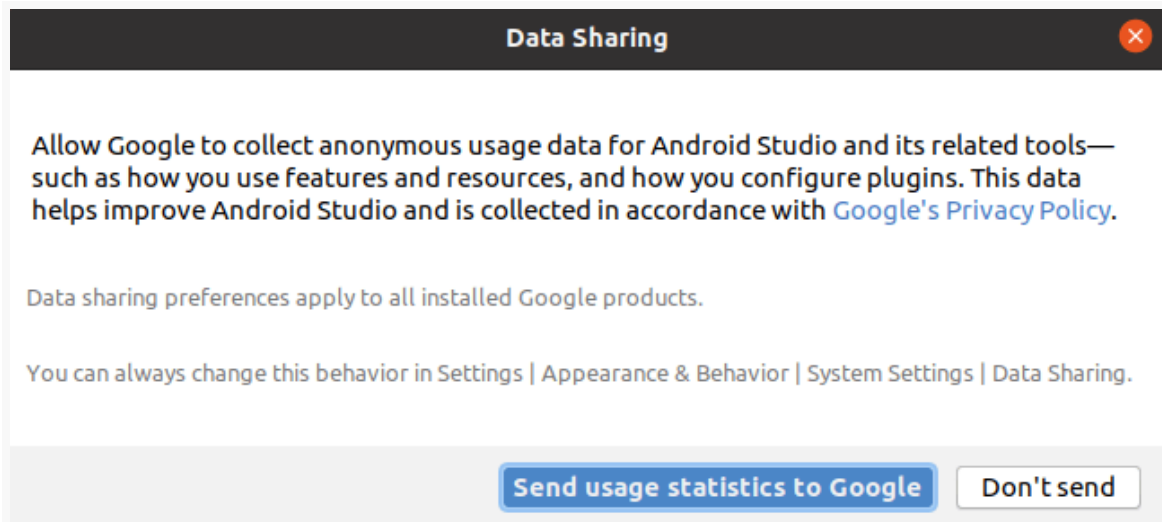
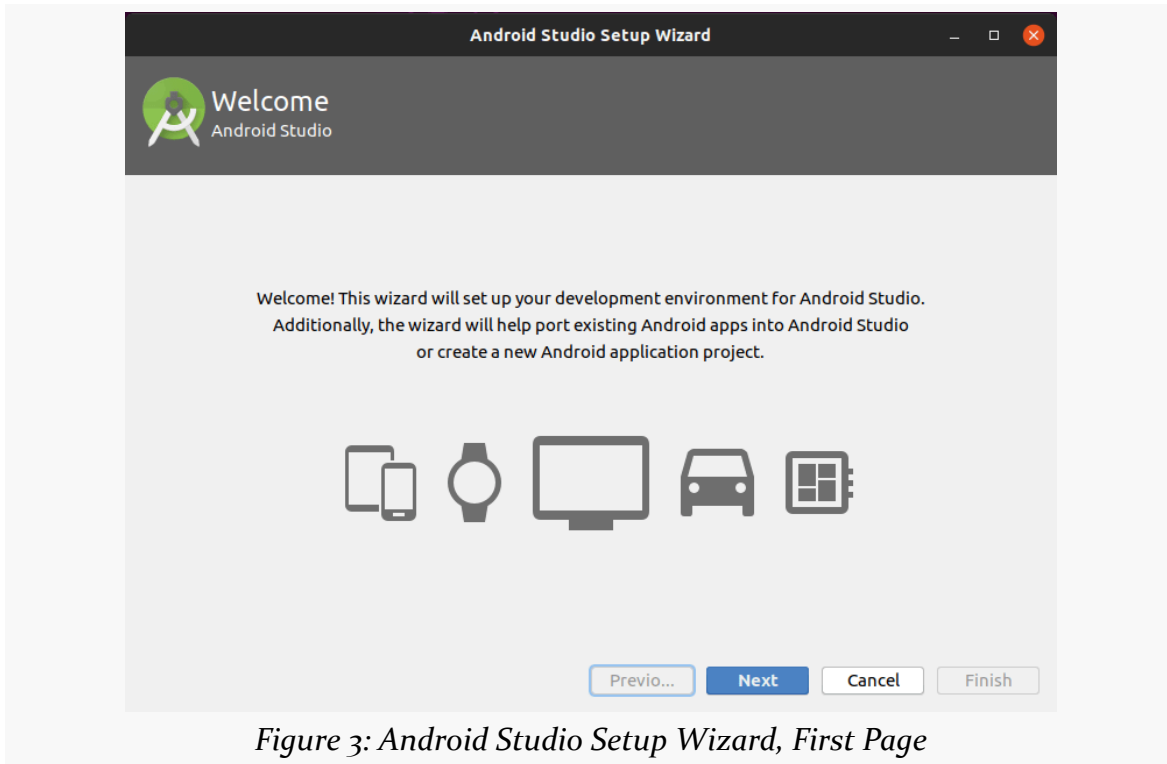


Figure 2: Android Studio Data Sharing Dialog

Click whichever button you wish.

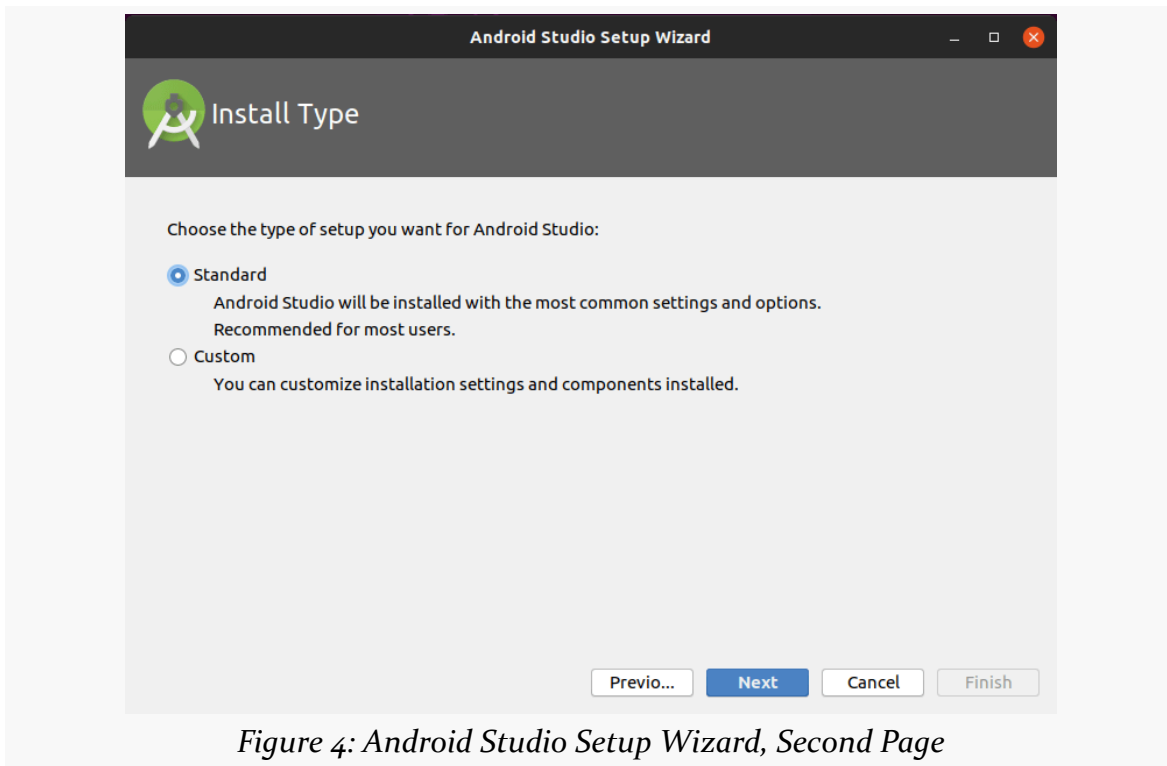
SETTING UP THE TOOLS

Eventually, you will be taken to the Android Studio Setup Wizard:



SETTING UP THE TOOLS

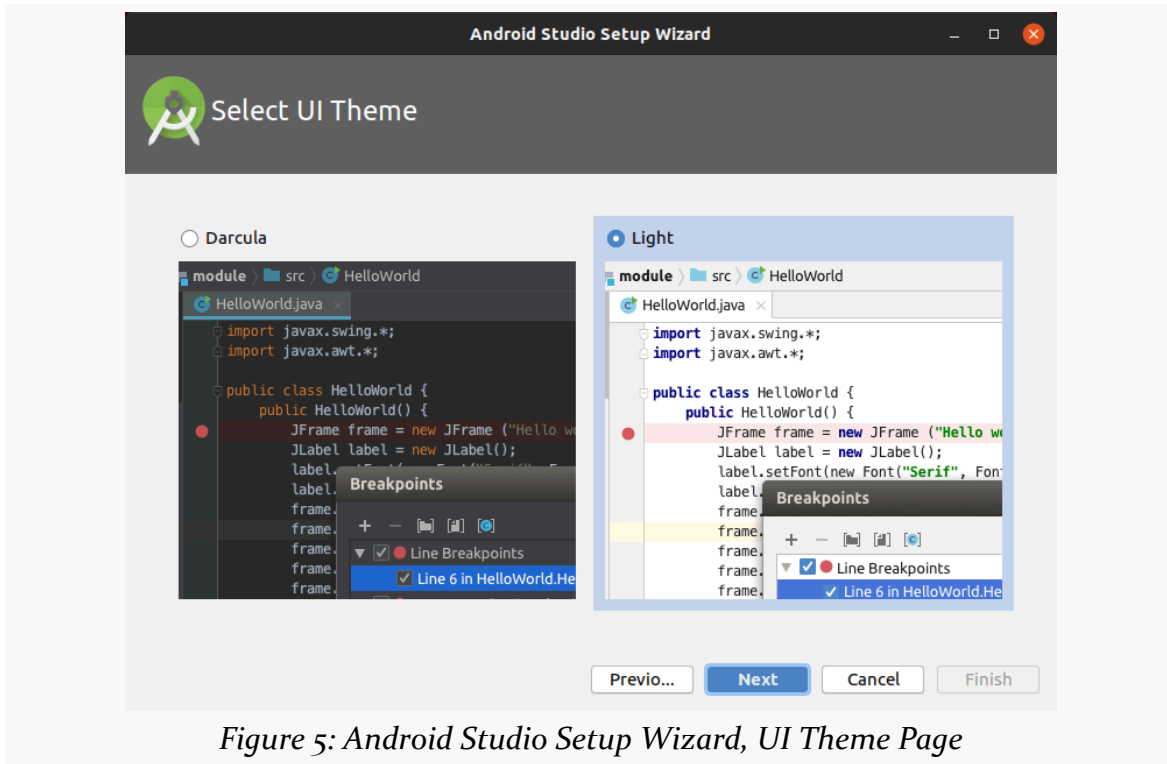
Just click “Next” to advance to the second page of the wizard:



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

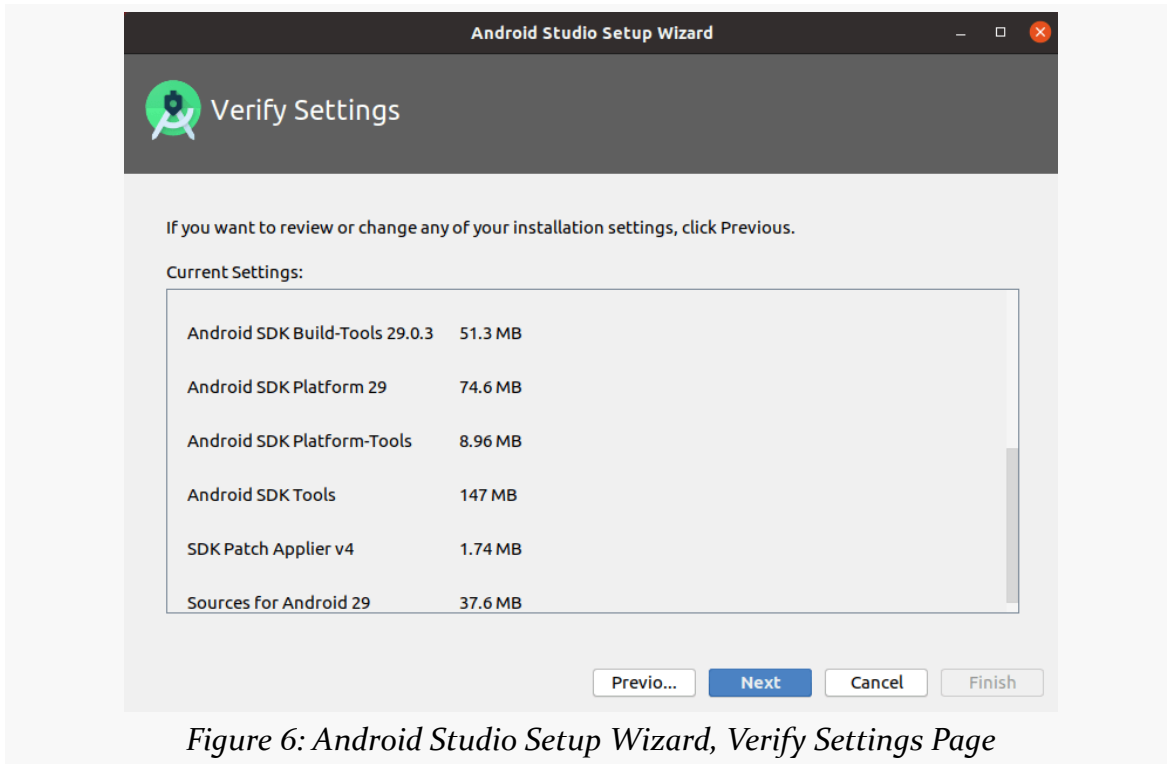
SETTING UP THE TOOLS

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



SETTING UP THE TOOLS

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



SETTING UP THE TOOLS

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

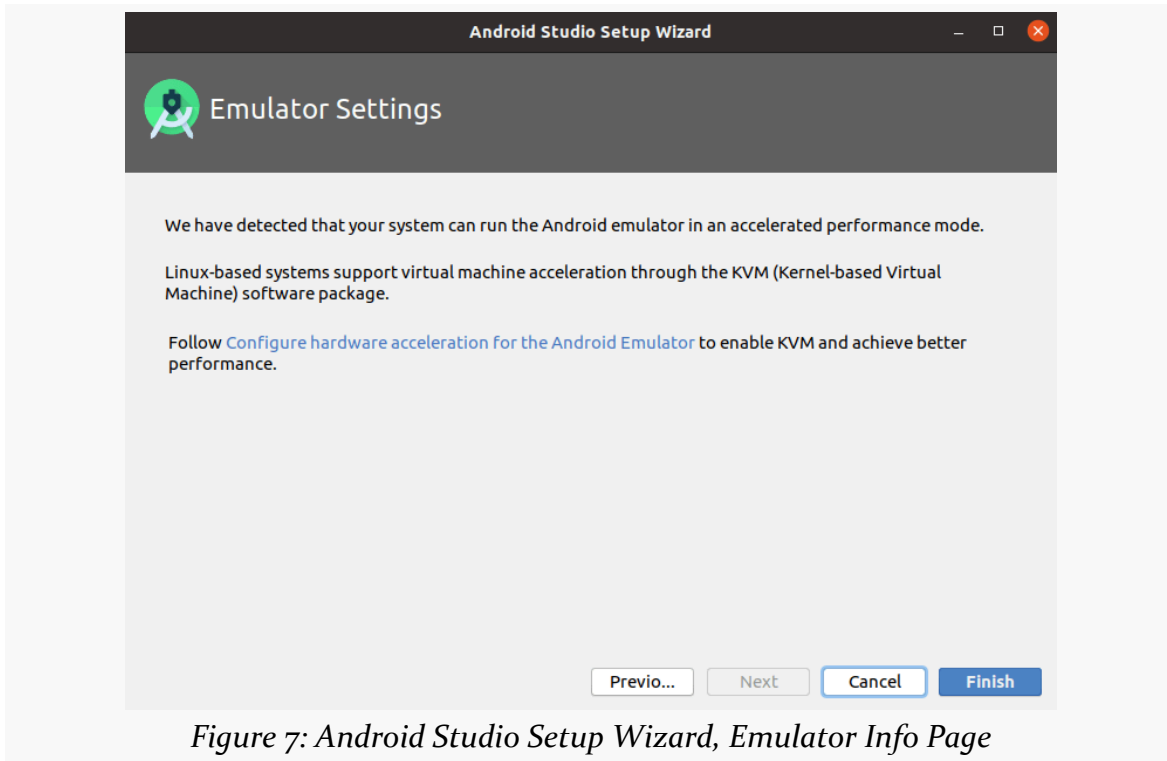


Figure 7: Android Studio Setup Wizard, Emulator Info Page

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, Android Studio will busily start downloading stuff to your development machine.

SETTING UP THE TOOLS

Clicking “Finish” when that is done will then take you to the Android Studio Welcome dialog:

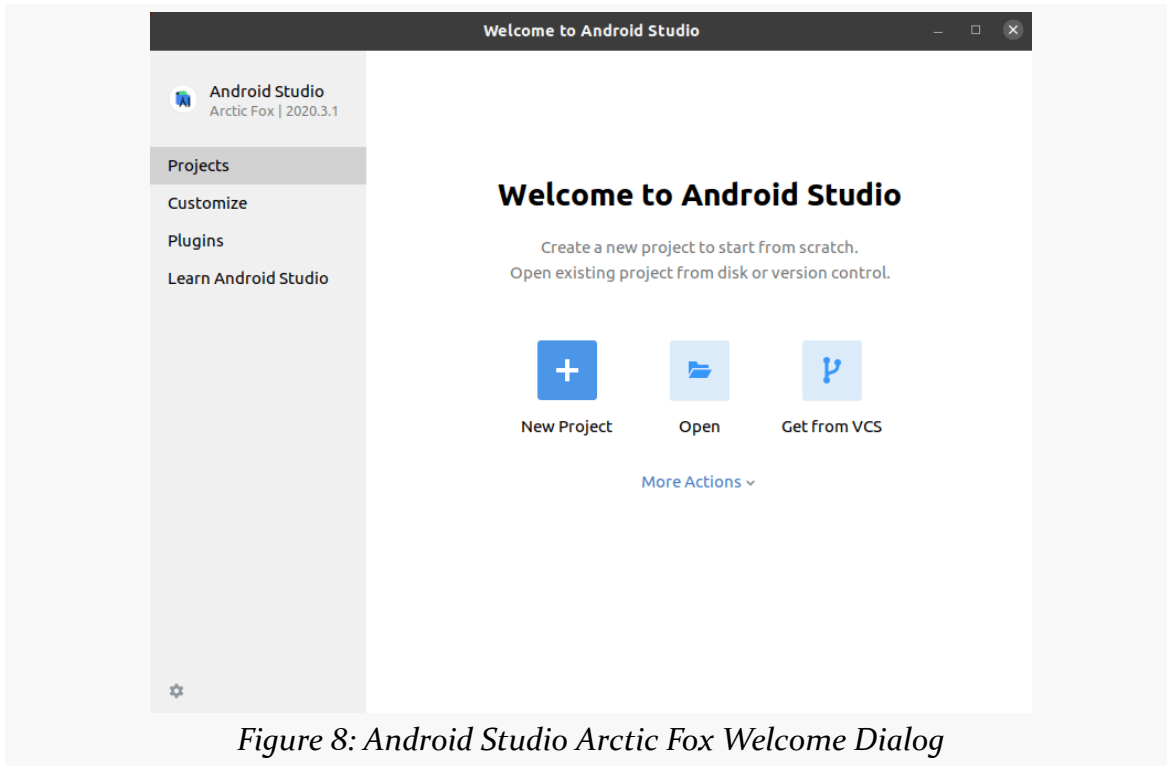


Figure 8: Android Studio Arctic Fox Welcome Dialog

Getting Your First Project

Creating an Android application first involves an Android “project”. As with many other development environments, the project is where your source code and other assets (such as icons) reside. And, the project contains the instructions for your tools for how to convert that source code and other assets into an Android app.

So, let’s work on setting up a “hello, world!” application to examine.

As with the rest of this book, these instructions are for Android Studio Arctic Fox. If you are using a different version of this IDE, while the instructions are likely to be close to what you will see, there will be differences.

Step #1: Importing the Project

Roughly speaking, there are two ways to start with a project with Android Studio: creating a new project or importing an existing project.

It might sound like creating a new project would be the more common scenario. In truth, many developers import an existing project, because they are working on a development team, and somebody else on the team created the project. Often, that project was created quite some time ago, with developers coming and going from the team.

So, while we will see how to create a project later in the book, for now, let’s import an existing project, one set up for use by this book. It will closely resemble the sort of project that you get when creating a brand-new project in Android Studio.

You can download this project from [the CommonsWare site](#). Then, unZIP that project to some place on your development machine. It will unZIP into an

GETTING YOUR FIRST PROJECT

HelloWorld/ directory.

Then, import the project. From the Android Studio welcome dialog — where we ended up at the end of the previous chapter — you can import a project via the “Import project (Eclipse ADT, Gradle, etc.)” option. If you already have a project open in Android Studio, you can import a project via File > New > Import Project... from the main menu.

Importing a project brings up a typical directory-picker dialog. Pick the HelloWorld/ 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 a mostly-empty Android Studio IDE window:

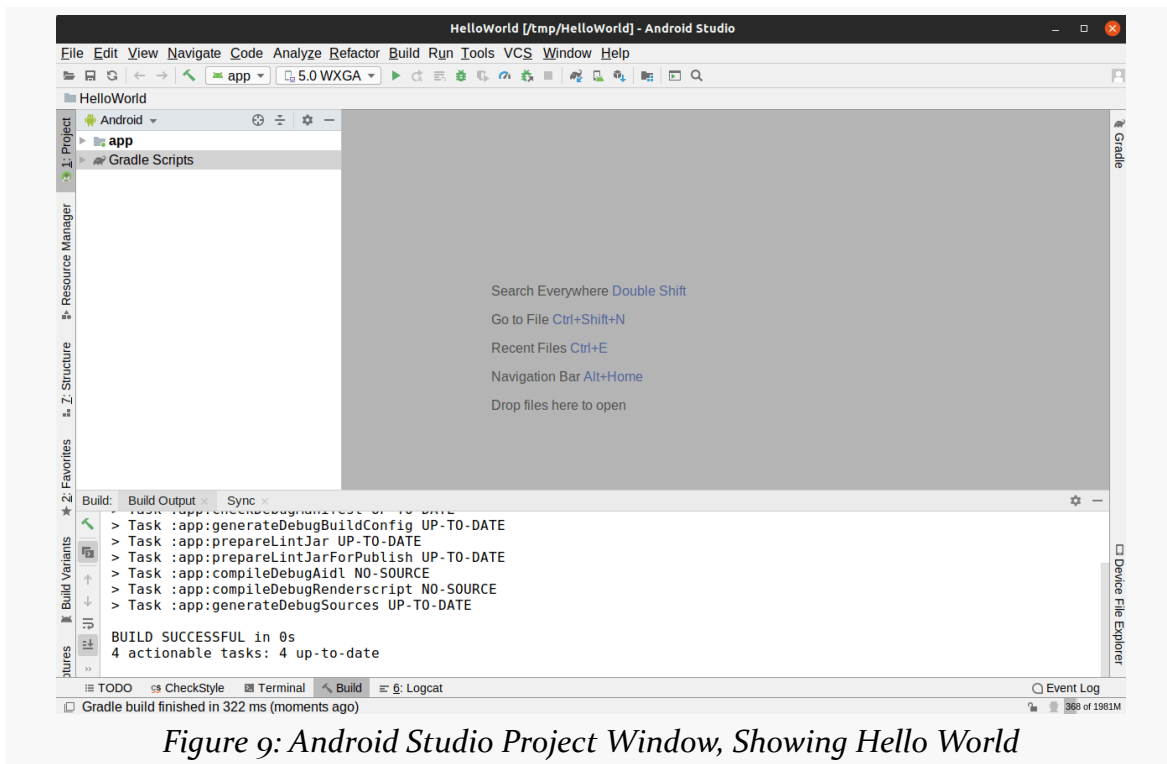


Figure 9: Android Studio Project Window, Showing Hello World

We will examine what is all in this window coming up in future chapters. But, first, let's get things set up for you to be able to run this sample app and see its results.

Step #2: Get Ready for the x86 Emulator

Your next 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](#).

Also, this book does not cover the Apple M1 CPU — you will need to determine what the current instructions are for getting a working emulator in that environment.

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 continue with [the next step](#).

If, on the other hand, you got some error messages in the installation wizard regarding HAXM, you would need to address those first. Unfortunately, there is so much variety in PC hardware and possible problems that this book cannot help you diagnose and fix your HAXM problems.

Mac

The wizards of Cupertino set up their Mac hardware to be able to run the Android x86 emulator without additional configuration. This is really 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](#).

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 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.

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

You should be taken to “welcome”-type screen:

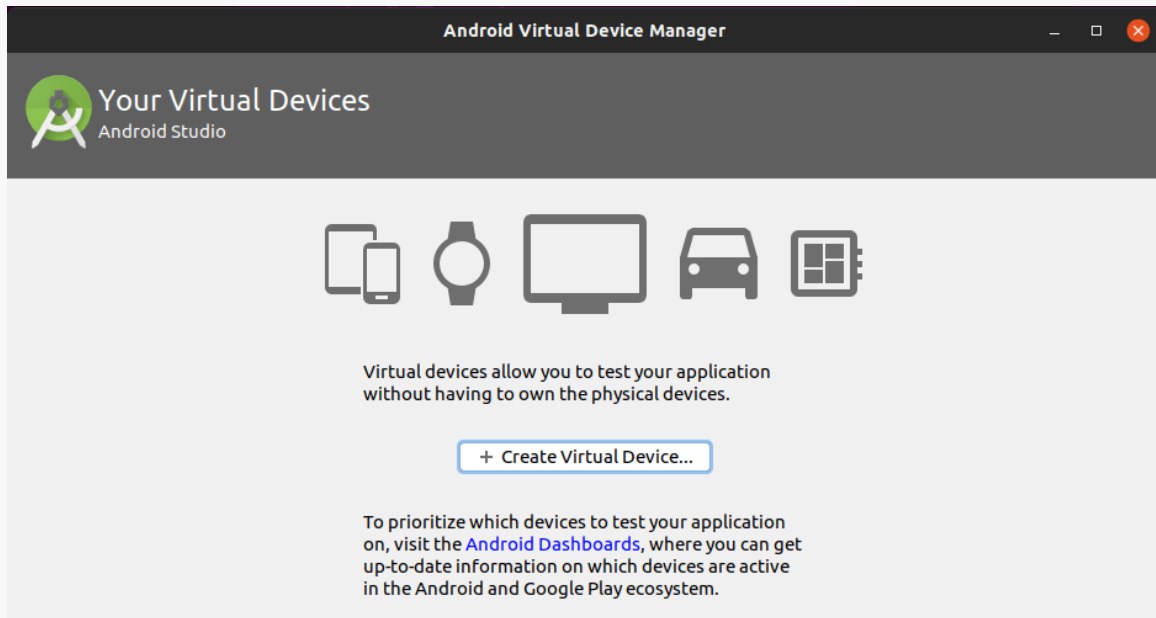
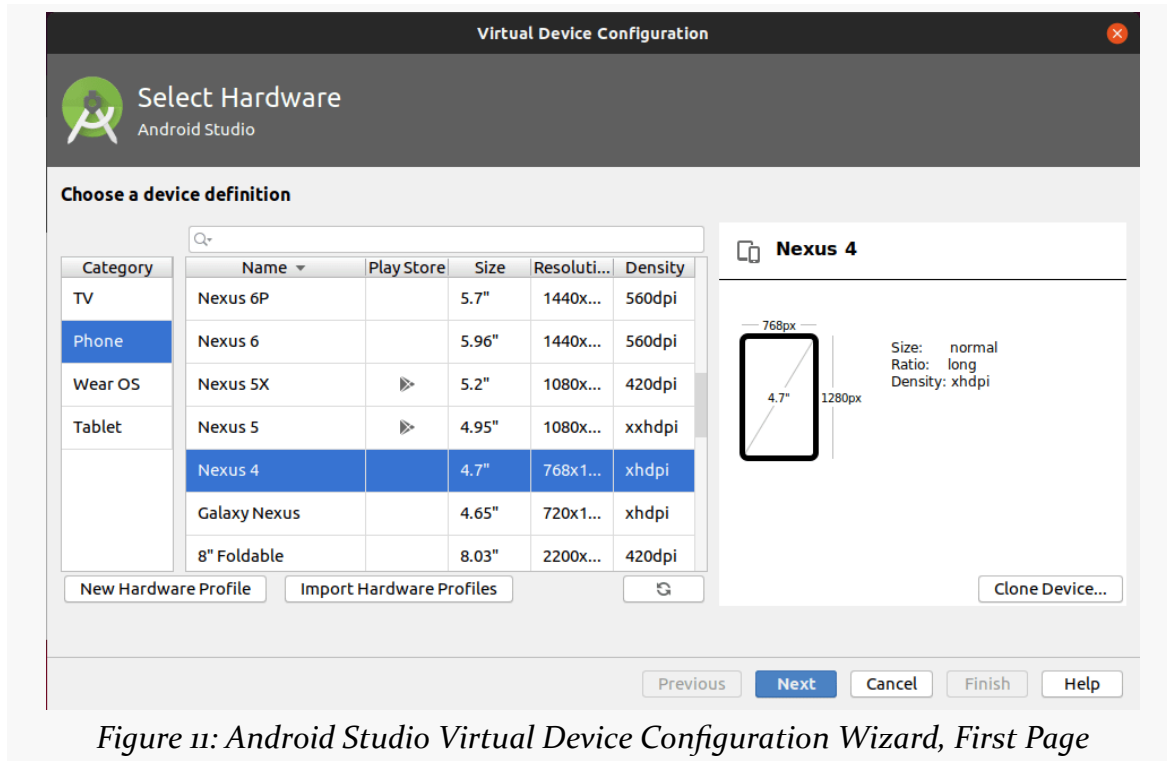


Figure 10: Android Studio AVD Manager, Welcome Screen

GETTING YOUR FIRST PROJECT

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



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, a 1280x768 or 1280x720 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 Nexus 4 image is a likely choice to start with.

GETTING YOUR FIRST PROJECT

Clicking “Next” allows you to choose an emulator image to use:

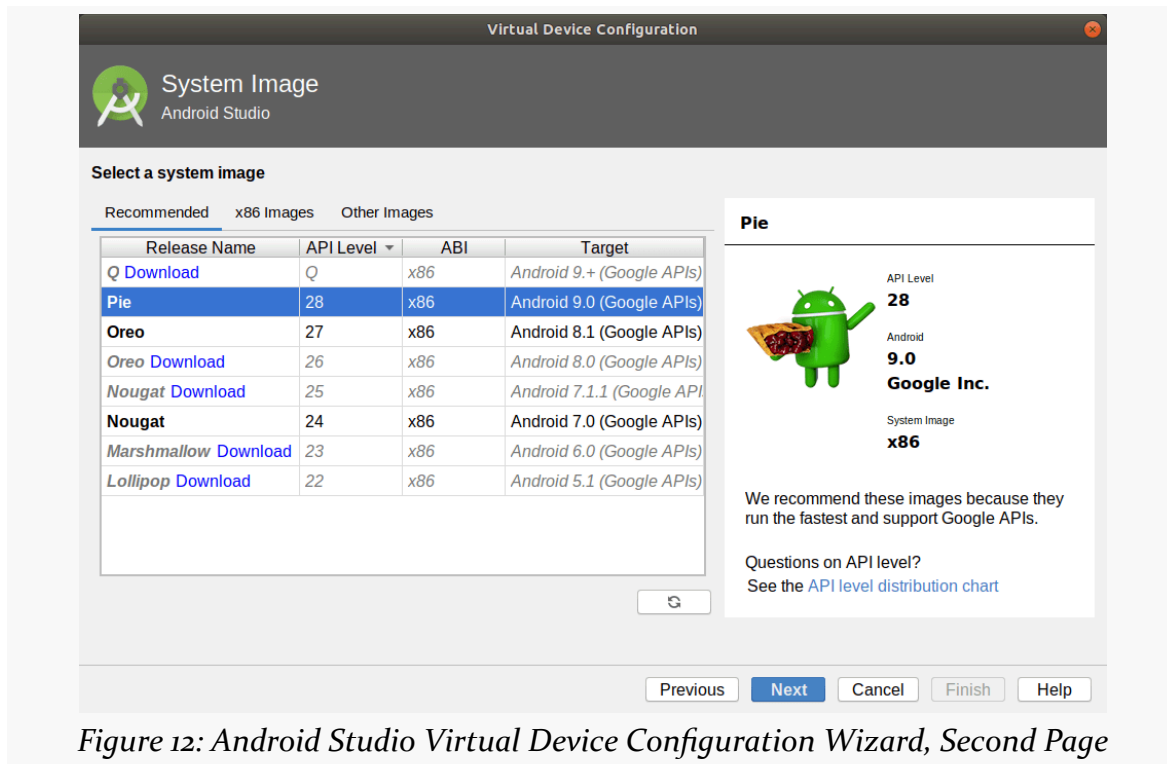


Figure 12: Android Studio Virtual Device Configuration Wizard, Second Page

The emulator images are spread across three tabs:

- “Recommended”
- “x86 Images”
- “Other Images”

Each of those tabs lists a bunch of possible emulator images, where those tables have cryptic columns like “API Level” and “Release Name”. We will get into what those are a bit later in the book. For right now, the key column is the “Target” column. This will tell you what version of Android the emulator emulates, such as “Android 8.1” or “Android 5.1”. For the time being, whether the “Target” has “(Google APIs)” or not does not matter very much.

GETTING YOUR FIRST PROJECT

On some of these tabs, you should see some entries with a “Download” link, and you might see others without it. 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:

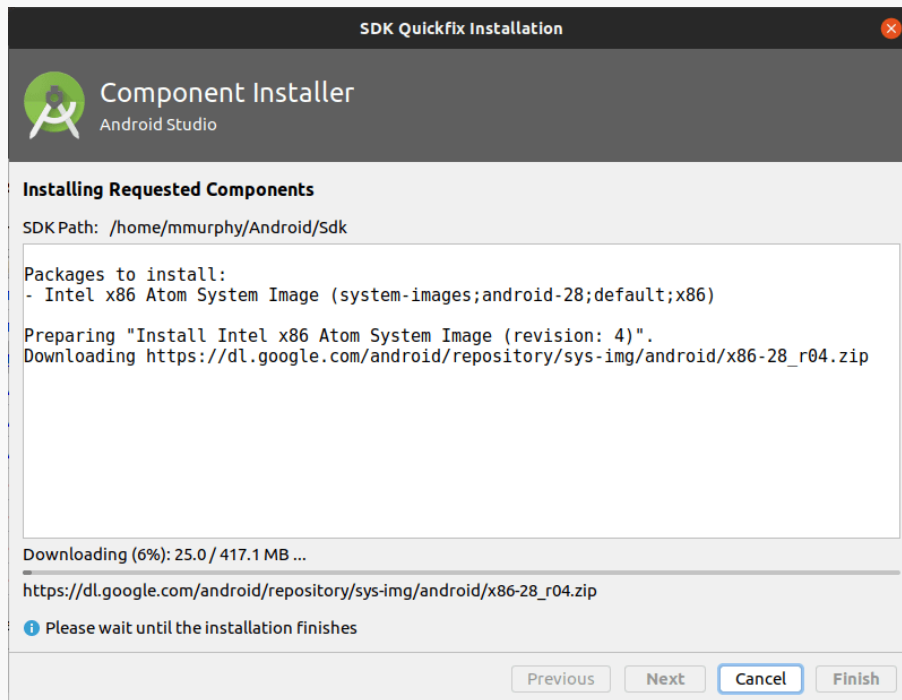


Figure 13: Android Studio Component Installer Dialog, Downloading API 28 Image

GETTING YOUR FIRST PROJECT

Once you have identified the image that you want — and have downloaded it if needed — click on one of them in the wizard. Clicking “Next” allows you to finalize the configuration of your AVD:

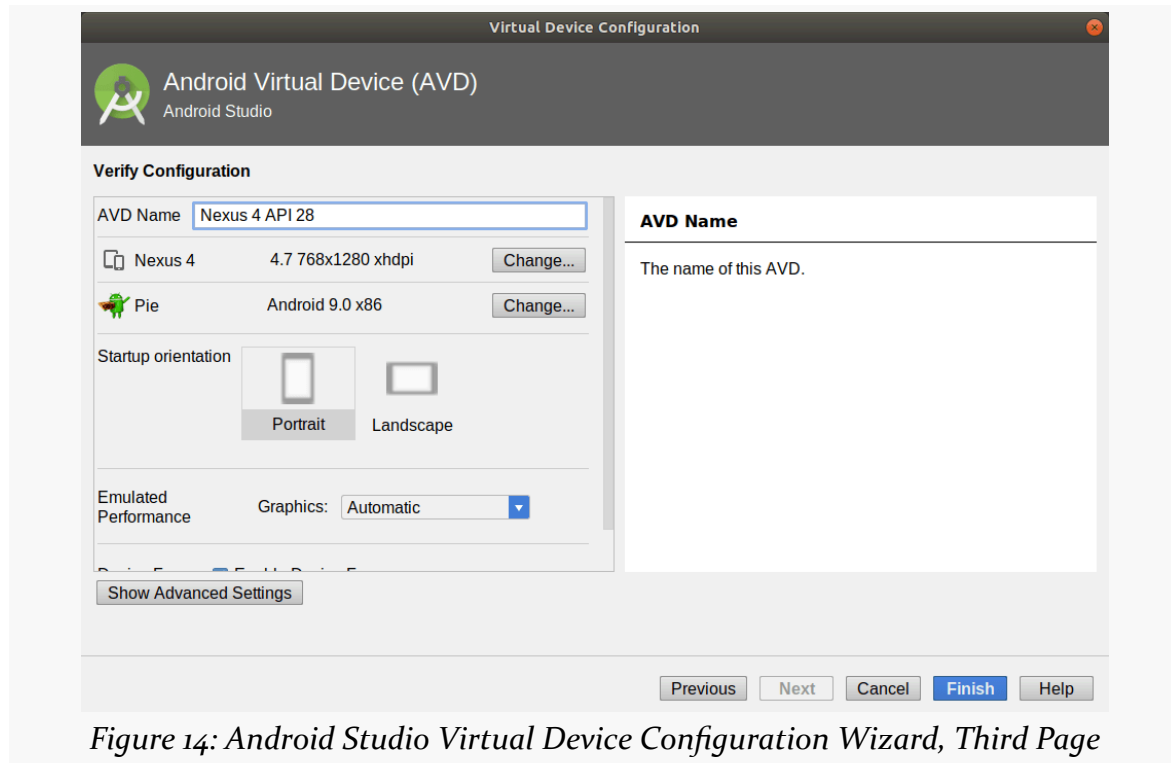


Figure 14: Android Studio Virtual Device Configuration Wizard, Third Page

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

GETTING YOUR FIRST PROJECT

If you do not have an Android device that you wish to set up for development, you can skip this step and [jump ahead to Step #5](#).

First, we need to enable USB debugging on the device. To do this, go into the Settings app.

You will need to find the “Build number” entry in here. Normally, that is on an “About” screen in Settings, though some devices have it in a separate screen (e.g., “Software Info”) off of the “About” screen.

Once you find the “Build number” entry, tap it seven times.

(yes, this is silly — just roll with it)

You should then see a brief popup message (a Toast) indicating that you are now a developer.

Then, you should have access to a “Developer options” item. Once again, the exact location of this varies by device, but usually it is either:

GETTING YOUR FIRST PROJECT

- An entry on the main Settings screen
- An entry in “System” > “Advanced options”, particularly on Android 8.0+ devices:

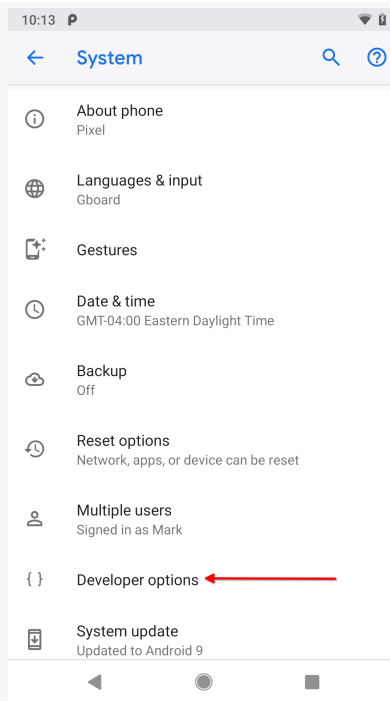


Figure 15: System Screen in Android 9.0 Settings App, Showing Advanced Options, with Developer Options Highlighted

GETTING YOUR FIRST PROJECT

Tapping on “Developer options” will bring up the Developer Options screen:

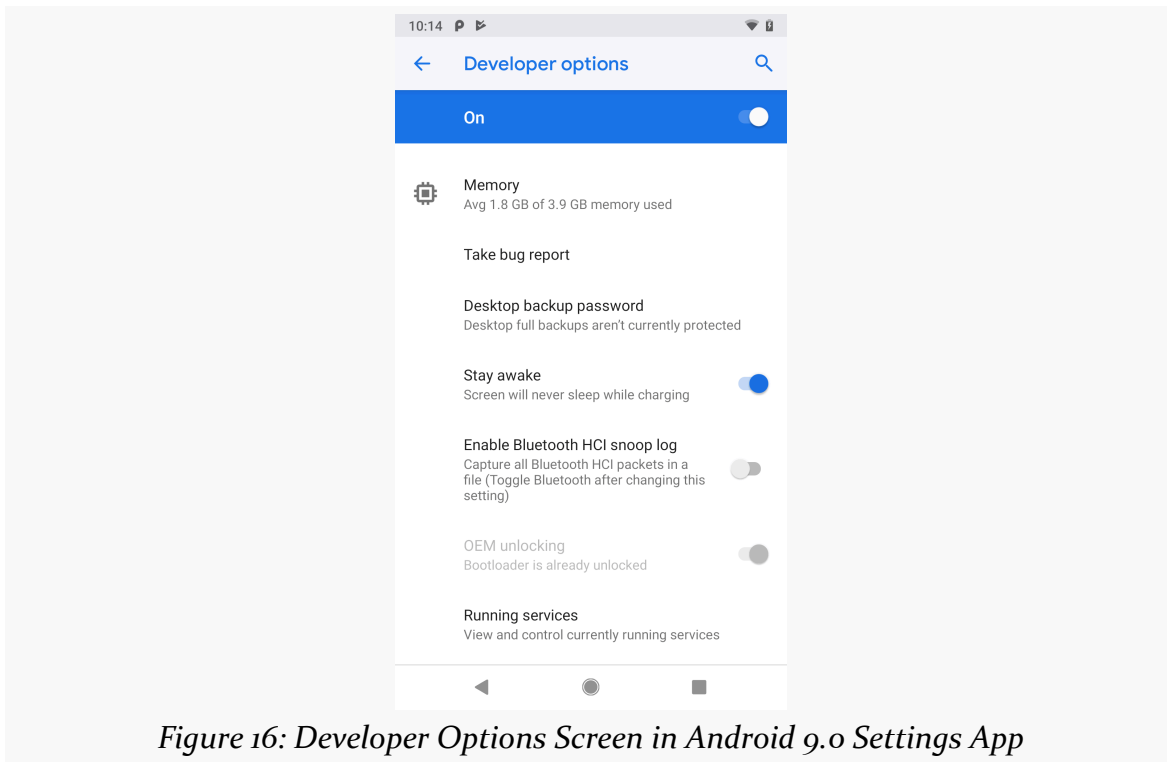
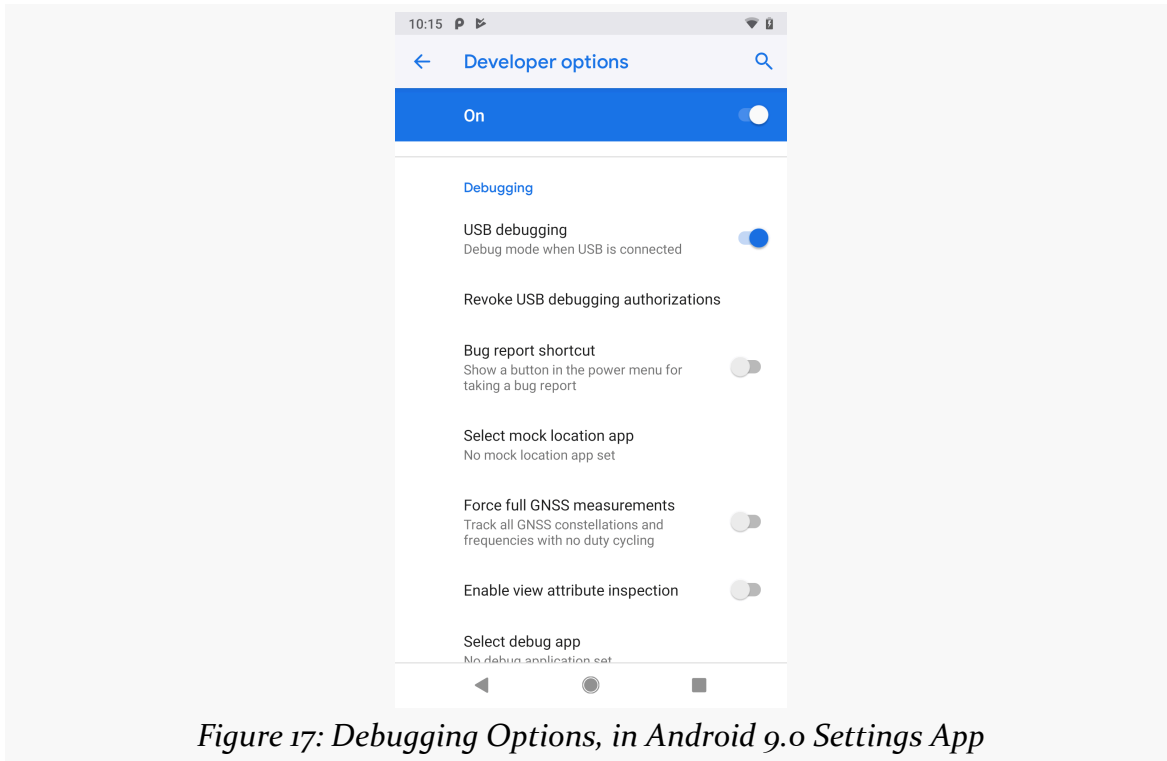


Figure 16: Developer Options Screen in Android 9.0 Settings App

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.

GETTING YOUR FIRST PROJECT

Then, scroll down and enable USB debugging, so you can use your device with the Android build tools:



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.

GETTING YOUR FIRST PROJECT

On devices running Android 4.2.2 or higher, 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:

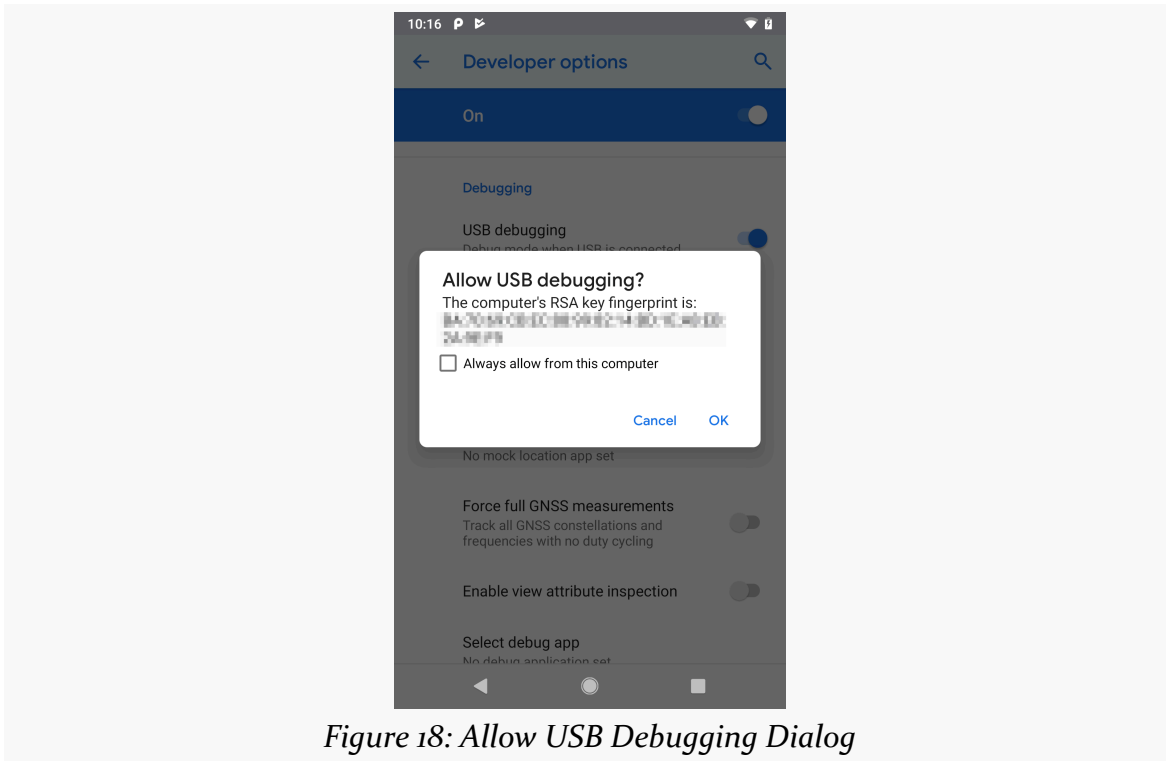


Figure 18: Allow USB Debugging Dialog

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, you can try to get one from the manufacturer links on [the Android Developer site](#).

macOS and Linux

It is likely that simply plugging in your device will “just work”.

If you are running Ubuntu (or perhaps other Linux variants), and when you later try running your app it appears that Android Studio does not “see” your device, you may need to add some udev rules. [This GitHub repository](#) contains some instructions and a large file showing the rules for devices from a variety of manufacturers, and [this blog post](#) provides more details of how to work with udev rules for Android devices.

Step #5: Running the Project

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

Android Studio has a toolbar just below the main menu. In that toolbar, you will find two drop-down lists, followed by the Run toolbar button (usually depicted as a green rightward-pointing triangle):



The first drop-down says “this is what I want to run”. Right now, your only viable option is “app”, referring to the app that this project builds.

The second drop-down says “this is where I want to run it”. Here, you will find a list of devices and emulators that are available to you.

GETTING YOUR FIRST PROJECT

To run the app, choose your desired device or emulator in the second drop-down, then click the Run toolbar button. If you choose an emulator, and the emulator is not already running, Android Studio will start it up. Then, after a short wait, your app should appear on it:



Figure 20: Android 9.0 Emulator with HelloWorld App

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:

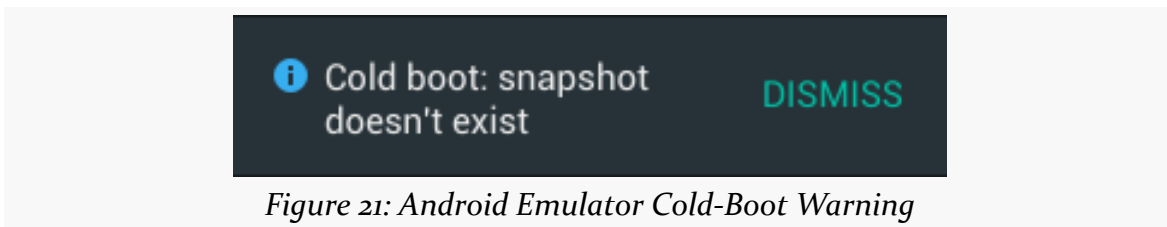
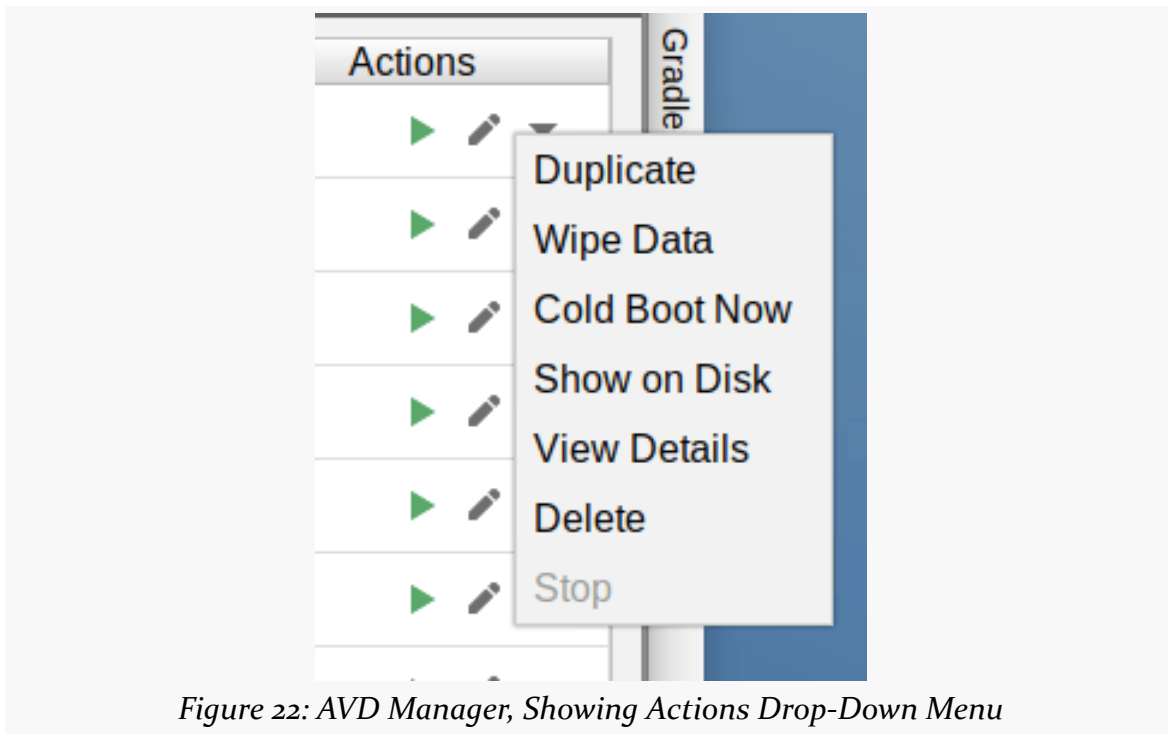


Figure 21: Android Emulator Cold-Boot Warning

The emulator behaves a bit like an Android device. Closing the emulator window is more like tapping the POWER button to turn off the screen. The next time you start

GETTING YOUR FIRST PROJECT

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”.



Taking a Tour of Android Studio

At this point, you have Android Studio set up, you have imported a project, and you have run that app on a device or emulator. Congratulations!

However, it may be useful for you to understand exactly what all of this does.

So, in this chapter and the next few that follow it, we are going to walk through what you set up in the previous two chapters, to explain what the pieces are and how they work together. We will start by examining Android Studio itself and the major things that you will be using.

The Project Tree

The “Project” view — docked by default on the left side, towards the top — brings up a way for you to view what is in the project.

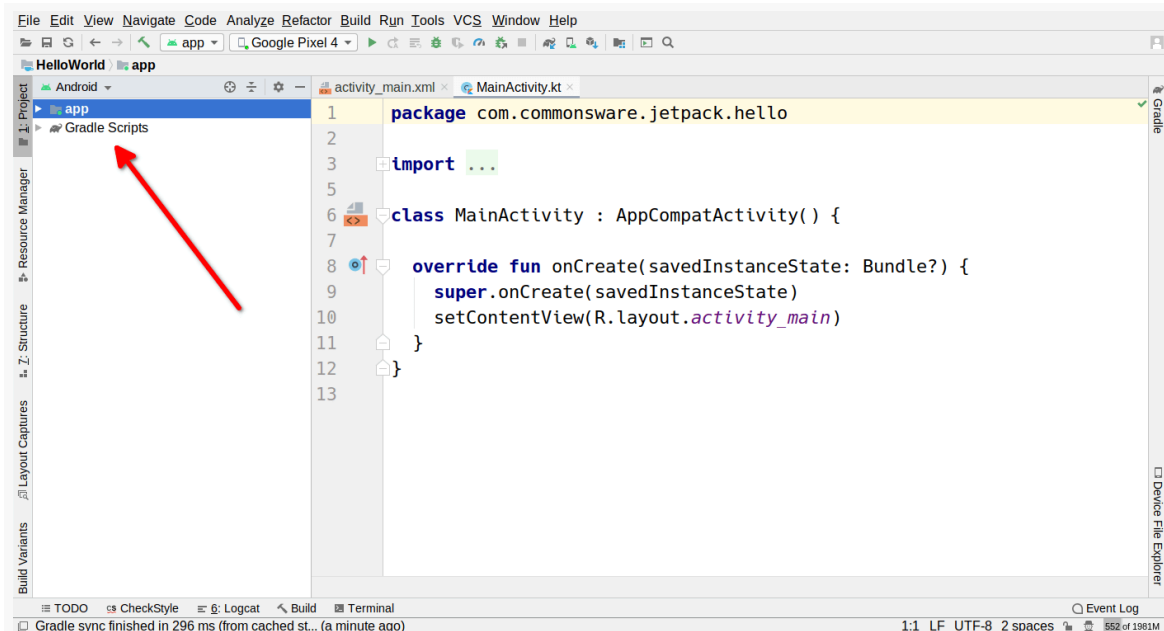


Figure 23: Android Studio Project View (Highlighted with Red Arrow)

What appears in this view is determined by the drop-down list above the tree itself:

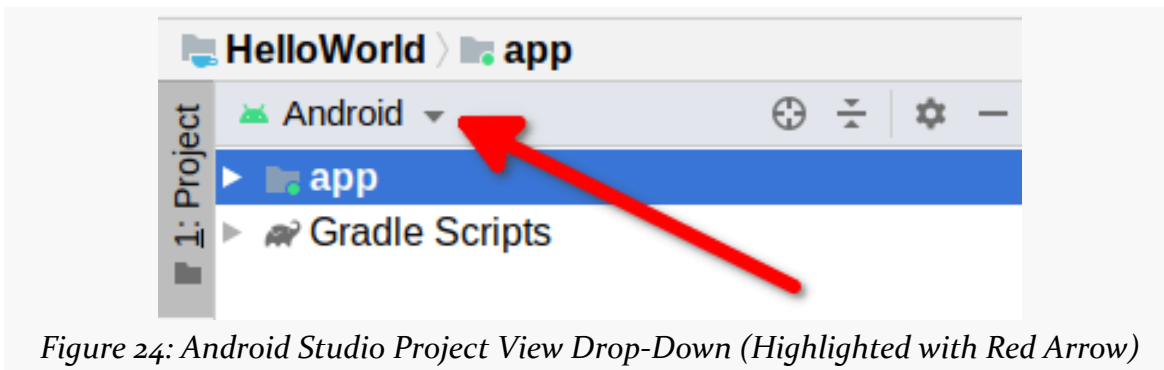


Figure 24: Android Studio Project View Drop-Down (Highlighted with Red Arrow)

TAKING A TOUR OF ANDROID STUDIO

The default is known as the “Android view”:

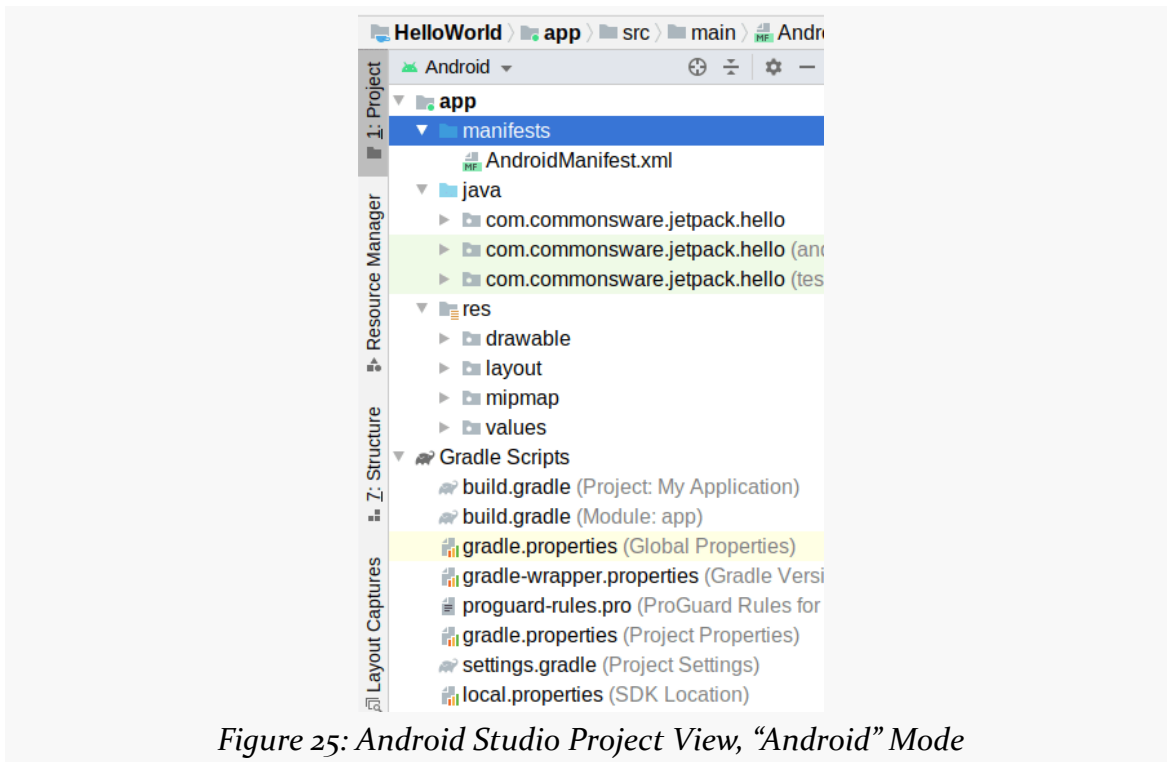


Figure 25: Android Studio Project View, “Android” Mode

TAKING A TOUR OF ANDROID STUDIO

You are welcome to use this if you wish. Many newcomers to Android are more comfortable changing that drop-down to be “Project”. This converts the tree to one showing the files that make up the project, with a typical sort of directories-and-files structure:

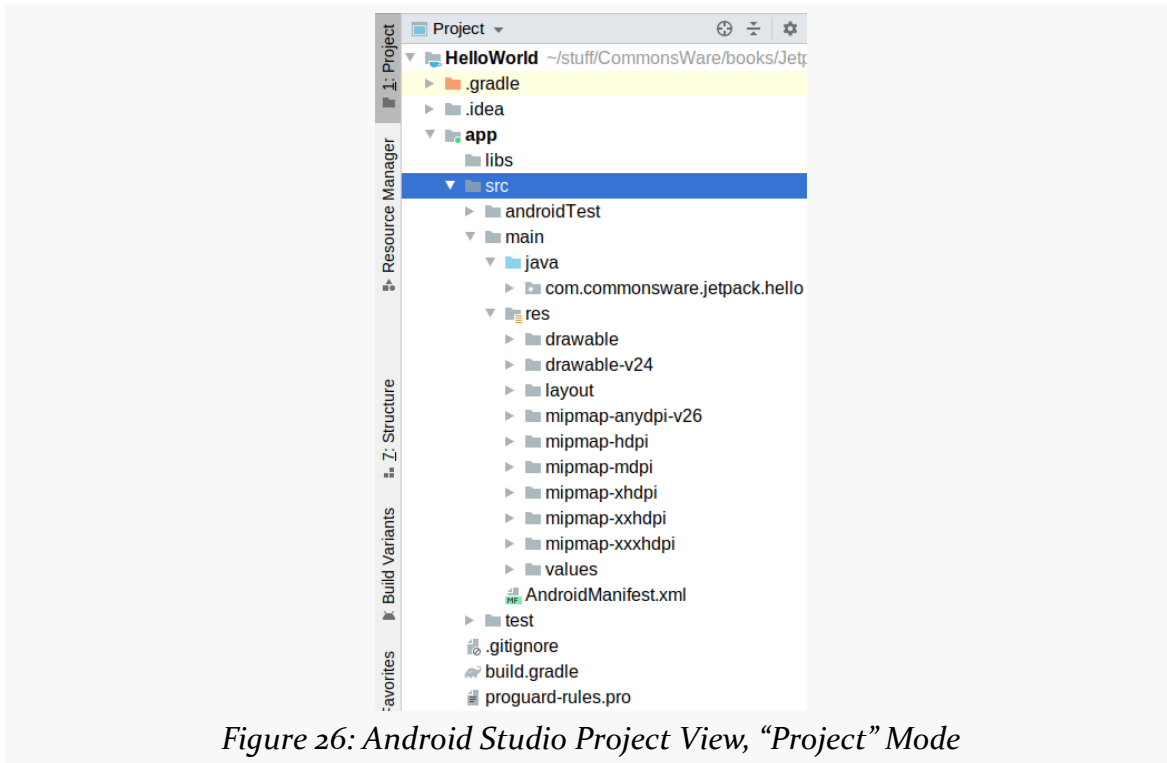


Figure 26: Android Studio Project View, “Project” Mode

This book usually will show the “Project” edition of the tree.

The Editing Pane

The biggest area of the IDE is devoted to the editing pane. Most files that you double-click on in the Project view will open in the editing pane. Each such file gets its own tab:

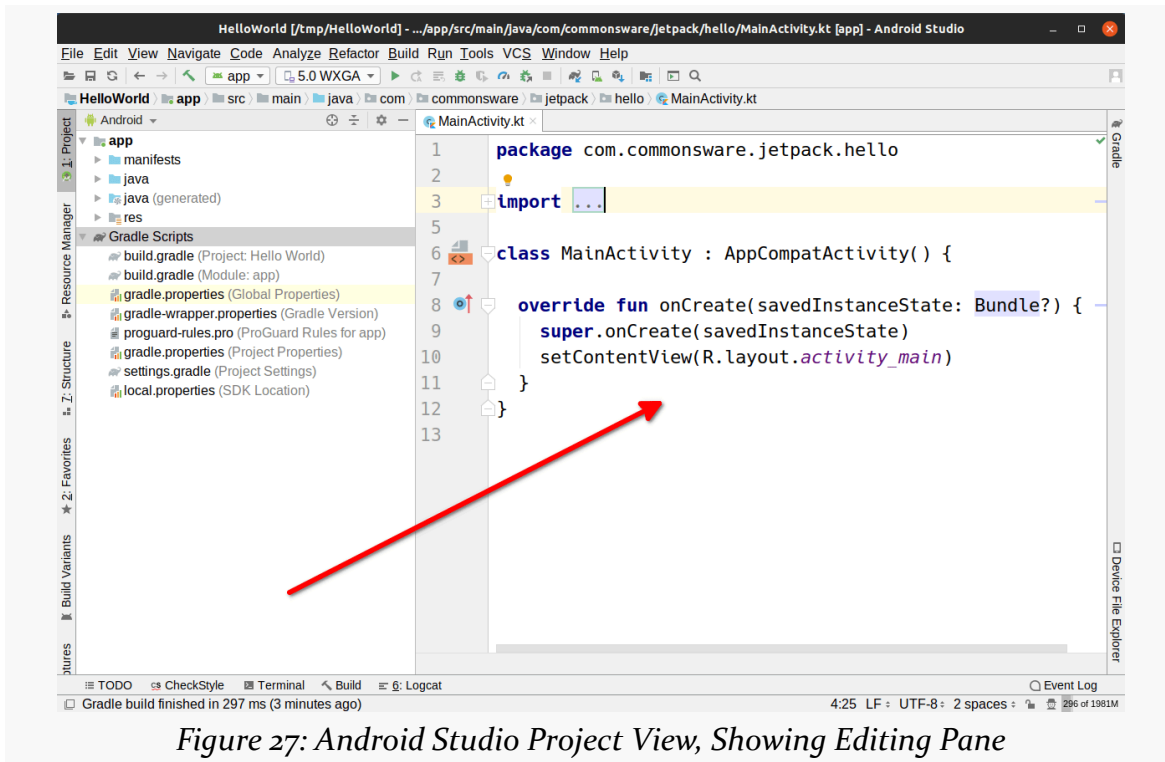


Figure 27: Android Studio Project View, Showing Editing Pane

The Docked Views

The “Project” view is not the only such view docked along the edges of the IDE. A variety of other such views are docked there by default, and the View > Tool Windows menu will offer other such views that you can display.

TAKING A TOUR OF ANDROID STUDIO

Some of these are “general purpose”, not strictly tied to Android app development. For example, the “Terminal” tool, docked by default towards the bottom of the IDE, brings up a terminal or command prompt, so you can execute command-line programs right from within the IDE:

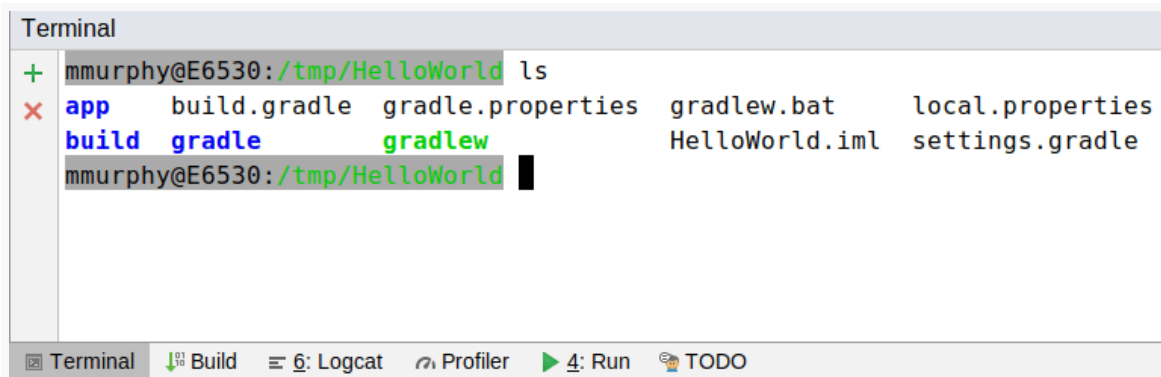


Figure 28: Android Studio Terminal

Others of these, such as “Gradle” and “Logcat”, are tied to Android app development, and we will examine those in greater detail as we explore different aspects of how to write Android apps.

Popular Menu and Toolbar Options

Across the top of the IDE are toolbars. These represent a subset of the items that are available in the IDE’s main menu. There are *lots* of toolbar options and *lots* of menu items. We will use some of these in the course of this book, but we will not be examining all of them.

By contrast, there are several menu items — many with corresponding toolbar buttons — that are fairly popular and are worth mentioning now.

We already used one of these toolbar buttons: the Run option, represented by a green triangle:



Figure 29: Android Studio Toolbar Segment

Additional Run Options

After you have run your app on a device or emulator, a few new toolbar buttons will show up, replacing the original Run button:

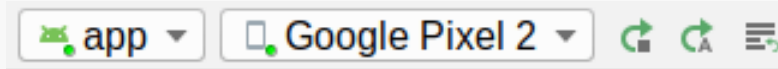


Figure 30: Post-Run Android Studio Toolbar Segment

The black square with a green curved arrow will re-run your app on the device or emulator. This will do the same thing as the Run button did back when it was just the simple green triangle.

The “A” with a green curved arrow, and the four lines with a tiny green curved arrow, will try to take changes that you have made in the IDE and simply “apply” them to the running copy of your app. For larger projects, this might be substantially faster than just running the app. However, it is somewhat risky — the resulting patched app might not be exactly the same as what you would get by just running it normally.

Debug

Near to those toolbar buttons is a large green bug:



Figure 31: Android Studio Debug Toolbar Button (Highlighted with Red Arrow)

This is the “Debug” button. By default, clicking this has much the same effect as clicking the Run button, other than Debug happening a bit more slowly. However, this runs your app under the control of the Android Studio debugger, where you can set breakpoints, inspect objects, and otherwise see what is going on when the app is running.

We will examine how to use the Android Studio debugger more [later in the book](#).

Open Project/Open Recent

In Android Studio, you can have several projects open at one time. Each project gets

TAKING A TOUR OF ANDROID STUDIO

its own separate window, with the same menus, toolbar, and so on as do the other windows.

To open another project, you can:

- Choose File > Open... from the main menu, to open an existing Android Studio project
- Choose File > Open Recent from the main menu, which will bring up a fly-out menu containing a list of projects that you have recently worked on, to be able to re-open those projects rapidly
- Choose File > New > Import Project from the main menu to import another project and set it up for use with Android Studio
- Choose File > New > New Project from the main menu to create a new project using a new-project wizard, which we will examine [later in the book](#)

To stop working on a project, just close its window. To stop Android Studio completely, close all of its windows. When you re-launch Android Studio, it will re-open the last project you had worked in, and you can get to other recent projects quickly via File > Open Recent.

AVD Manager

We saw how to set up an emulator with the AVD Manager in [the previous chapter](#).

To return to the AVD Manager, you can use the Tools > AVD Manager main menu option that you did before. Also, there is a toolbar button for more rapid access to the AVD Manager:

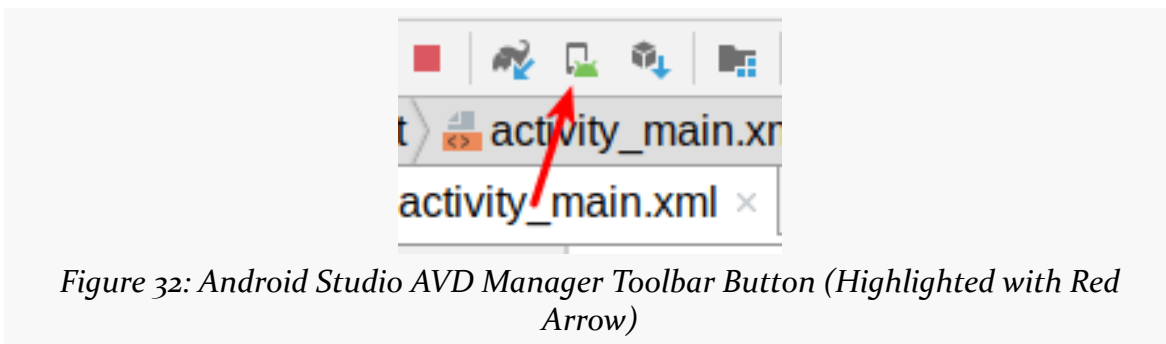


Figure 32: Android Studio AVD Manager Toolbar Button (Highlighted with Red Arrow)

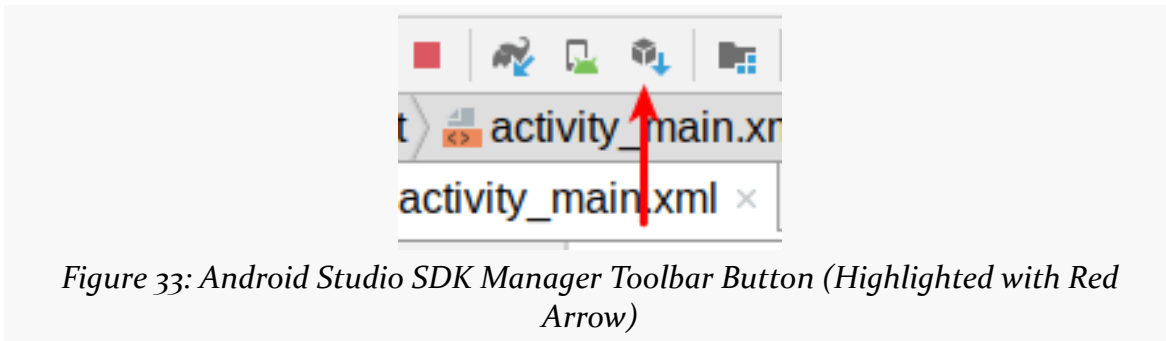
We will explore the AVD Manager, and working with the emulator, in greater detail in [an upcoming chapter](#).

SDK Manager

When you installed Android Studio and ran it for the first time, a lot of the tools, libraries, and related materials that form “the Android SDK” were also downloaded and installed for you.

When updates to Android Studio or installed pieces of the Android SDK are available, you will be prompted with a dialog or other form of pop-up, where you can elect to allow Android Studio to install the update. Note that installing updates may take a few minutes, depending on your Internet connection speed. Also note that installing updates may require you to restart Android Studio afterwards to apply those updates.

On occasion, you may find instructions telling you to go to “the SDK Manager” to install something. You can get to the SDK Manager via Tools > SDK Manager in the main menu or via its corresponding toolbar button:



We will explore the SDK Manager in greater detail [later in the book](#).

Settings

To control the overall behavior of Android Studio, there is a Settings dialog that you can display via “File” > “Settings” (on Windows and Linux; macOS has an equivalent option in “Android Studio” > “Preferences...”):

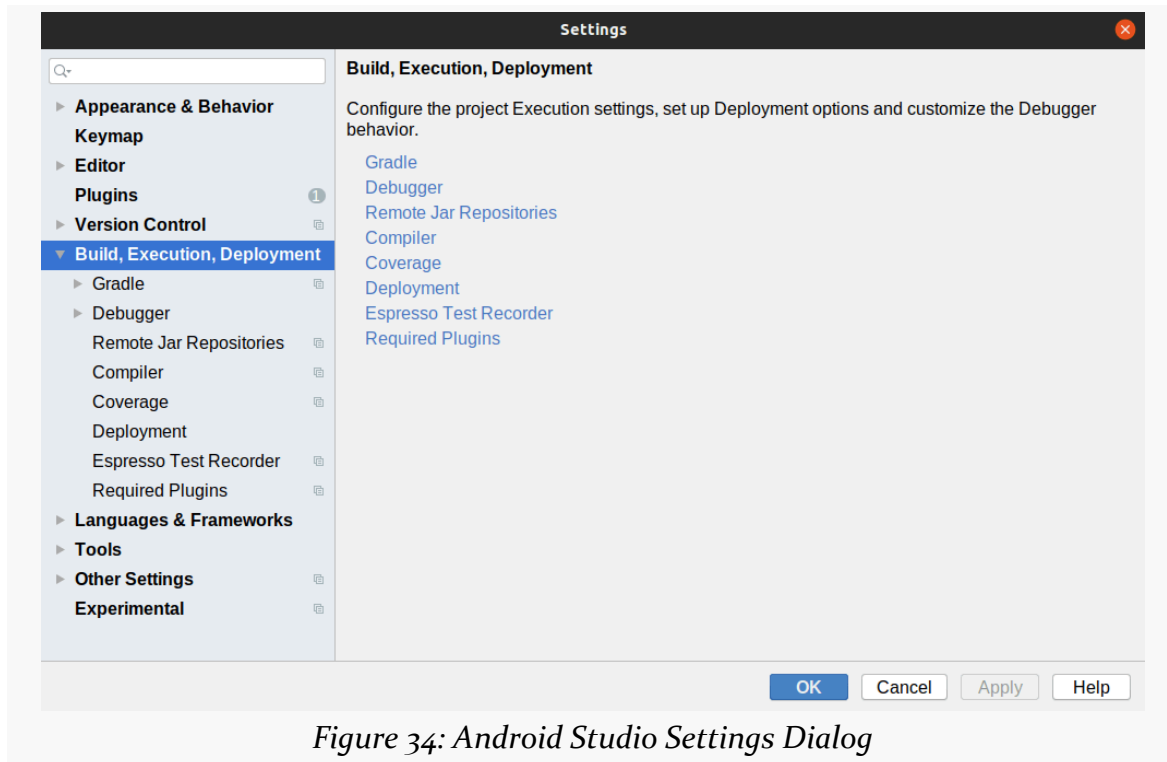


Figure 34: Android Studio Settings Dialog

There are a *lot* of settings that you can configure in the Settings dialog. You can either navigate via the tree on the left or via the search field.

Some popular things to tailor include:

TAKING A TOUR OF ANDROID STUDIO

- The color scheme to use for the IDE:

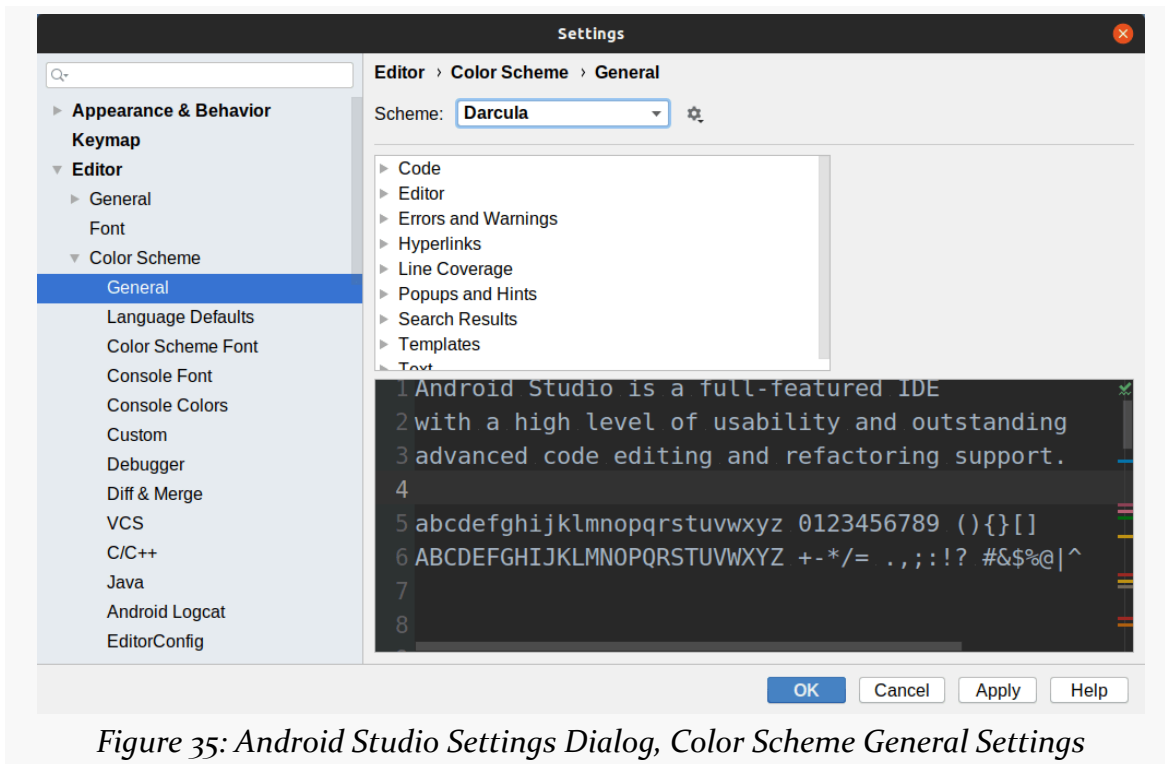


Figure 35: Android Studio Settings Dialog, Color Scheme General Settings

TAKING A TOUR OF ANDROID STUDIO

- The font and font size to use in the editing pane:

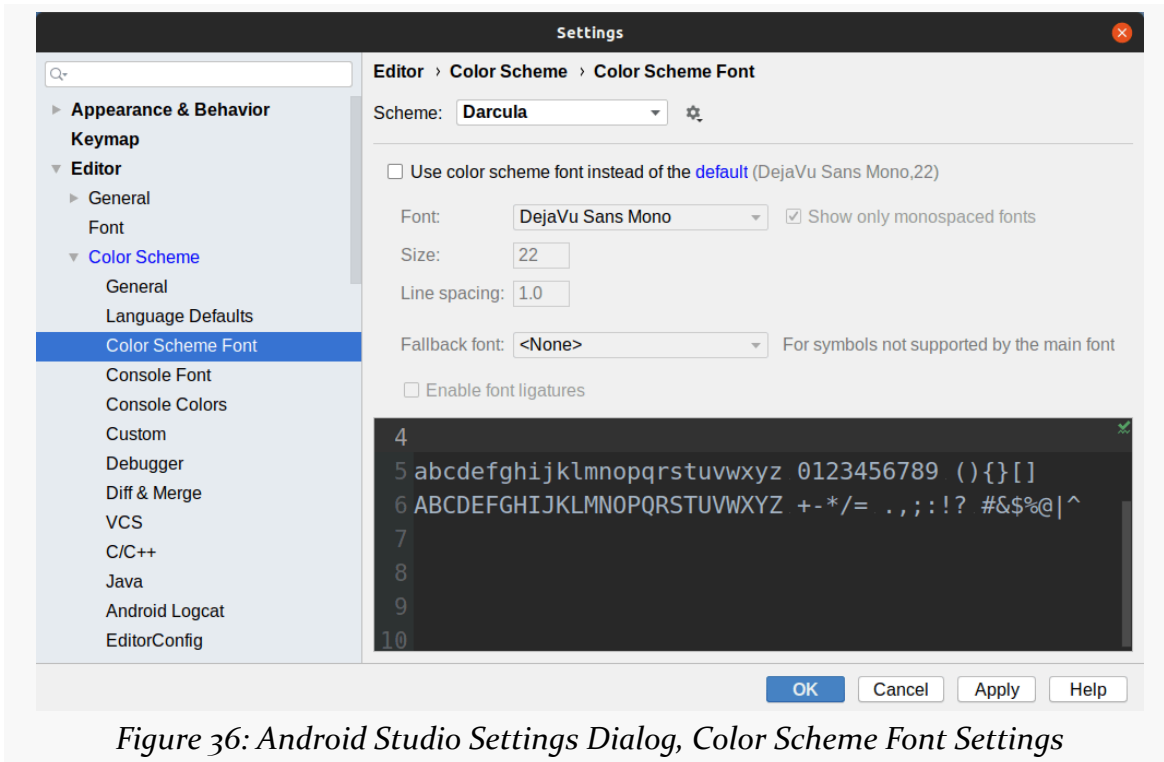


Figure 36: Android Studio Settings Dialog, Color Scheme Font Settings

TAKING A TOUR OF ANDROID STUDIO

- The coding style rules to apply to your source code, for languages like Java and Kotlin:

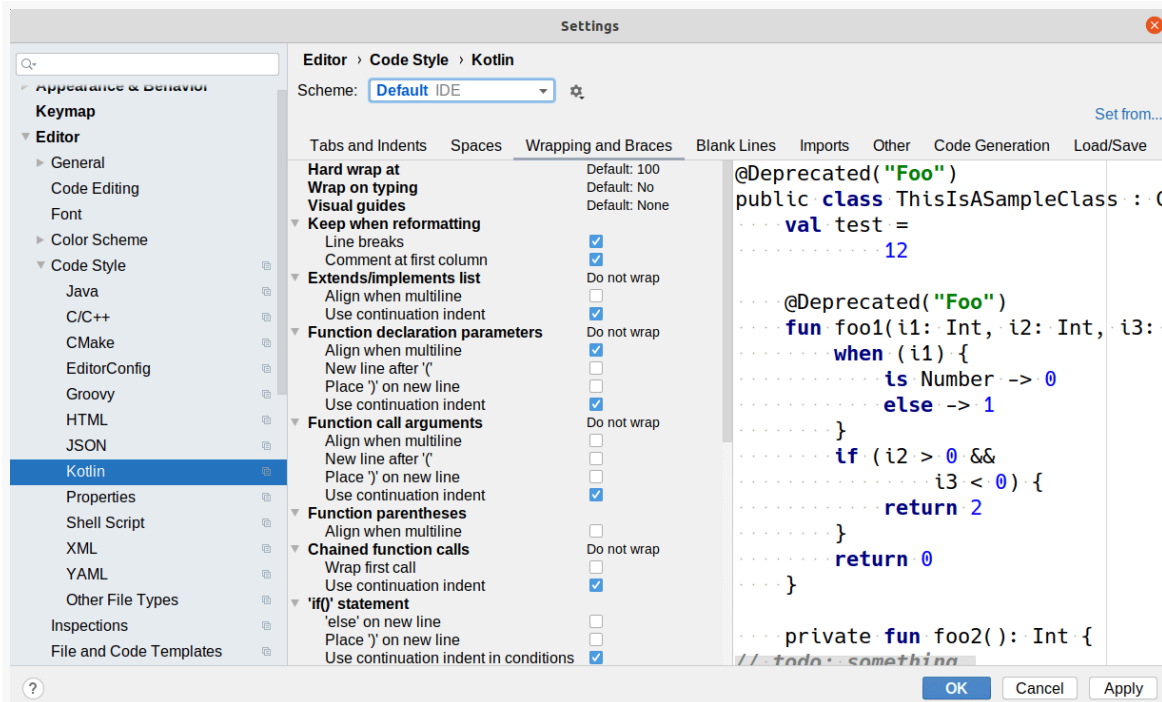


Figure 37: Android Studio Settings Dialog, Kotlin Code Style Settings

We will explore more options [later in the book](#).

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, allowing you to view release notes, and encouraging you to apply the update. If you choose the latter, the dialog downloads the update and restarts the IDE, applying the update along the way.

TAKING A TOUR OF ANDROID STUDIO

To control which channel's worth of updates you are getting, go to the Settings dialog, and in there go to "Appearance & Behavior > System Settings > Updates":

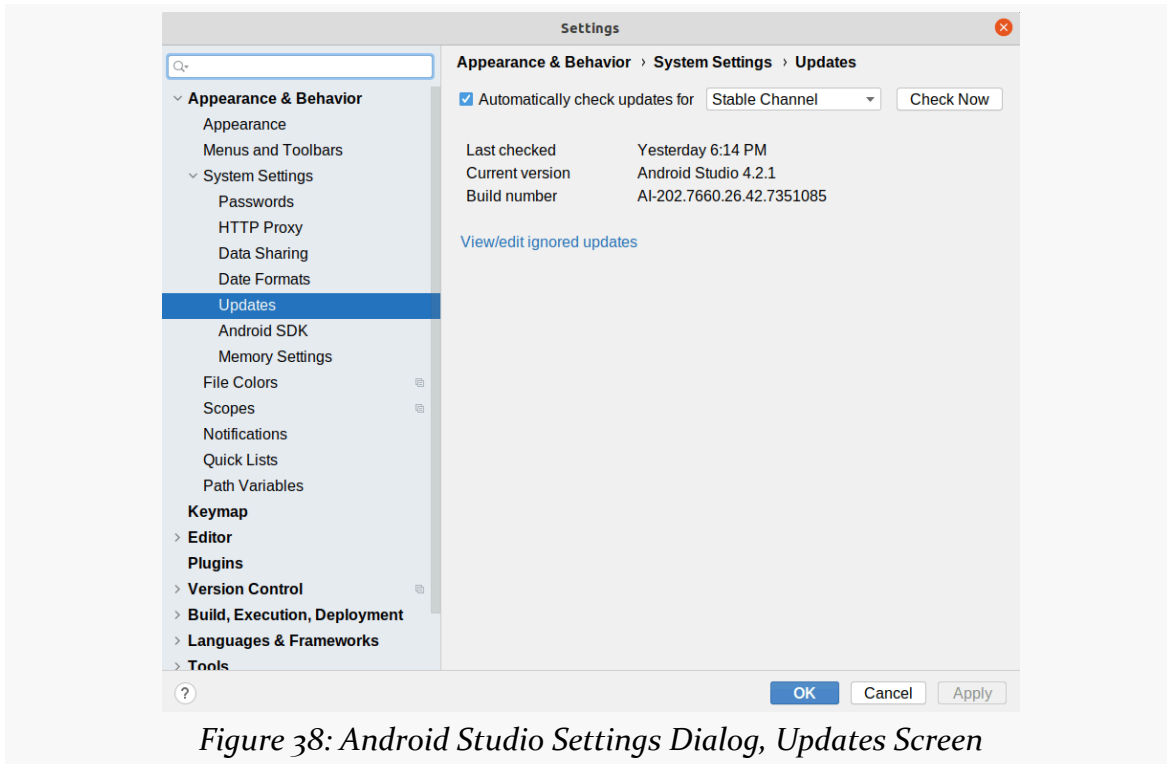


Figure 38: Android Studio Settings Dialog, Updates Screen

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)

For most developers, Stable is the best choice. Power users might consider one of the other channels.

TAKING A TOUR OF ANDROID STUDIO

When an update is available, Android Studio will tell you via a dialog:

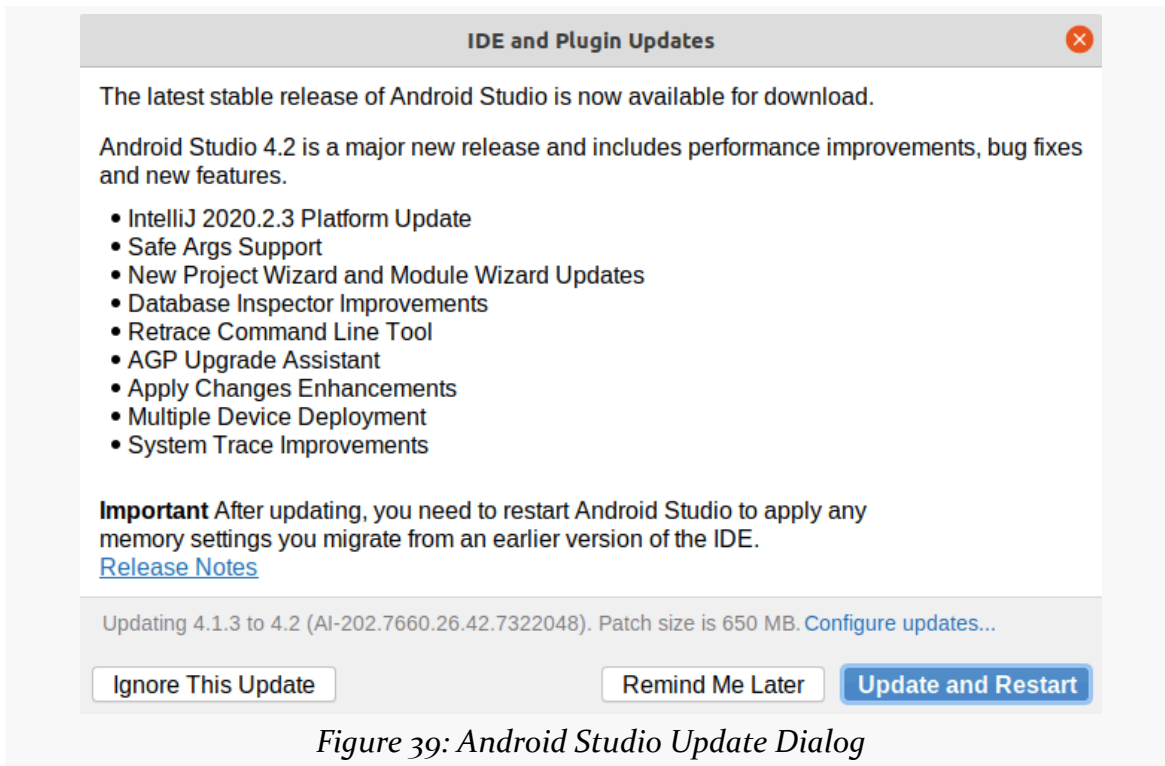


Figure 39: Android Studio Update Dialog

“Remind Me Later” will pop up the dialog in the future, while “Update and Restart” will apply the upgrade now and restart the IDE after upgrading it. “Ignore This Update” will stop the dialog from appearing automatically, but it will not apply the update... and usually you want the updates.

Examining Your Code

When you decided to learn how to write Android apps, most likely you were thinking about traditional computer programming, using programming languages like Java and Kotlin. There is a fair amount of such programming involved in Android apps, though perhaps less than you might think.

In this chapter, we will explore what our starter project contains in terms of the code and how that code is organized.

The Top Level

Let's look at our starter project's tree, as shown in the “Project” view (in the “Project” mode in the drop-down), focusing on the top level of entries:

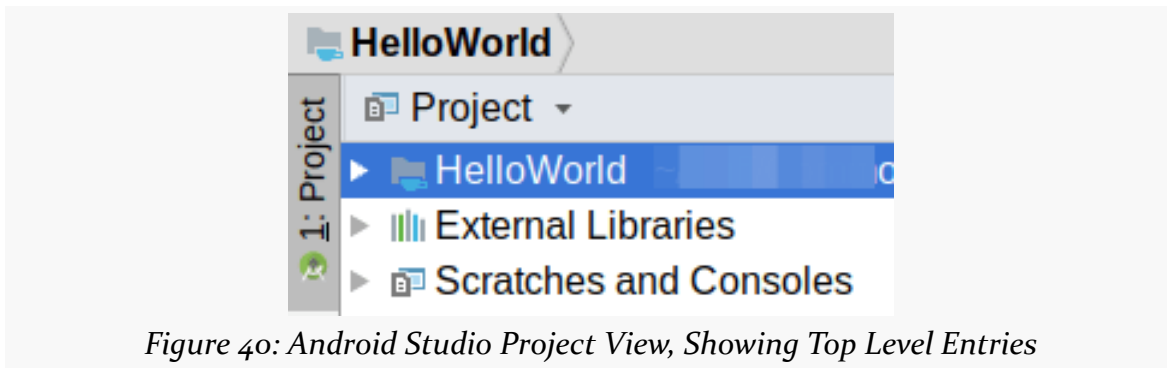


Figure 40: Android Studio Project View, Showing Top Level Entries

You will spend the vast majority of your time in the HelloWorld/ portion of the project tree, which represents the files that make up your Android app. We will examine the other two items — “External Libraries” and “Scratches and Consoles” — a bit [later in this chapter](#).

The Project Contents

That HelloWorld/ entry contains a fair number of files and subdirectories:

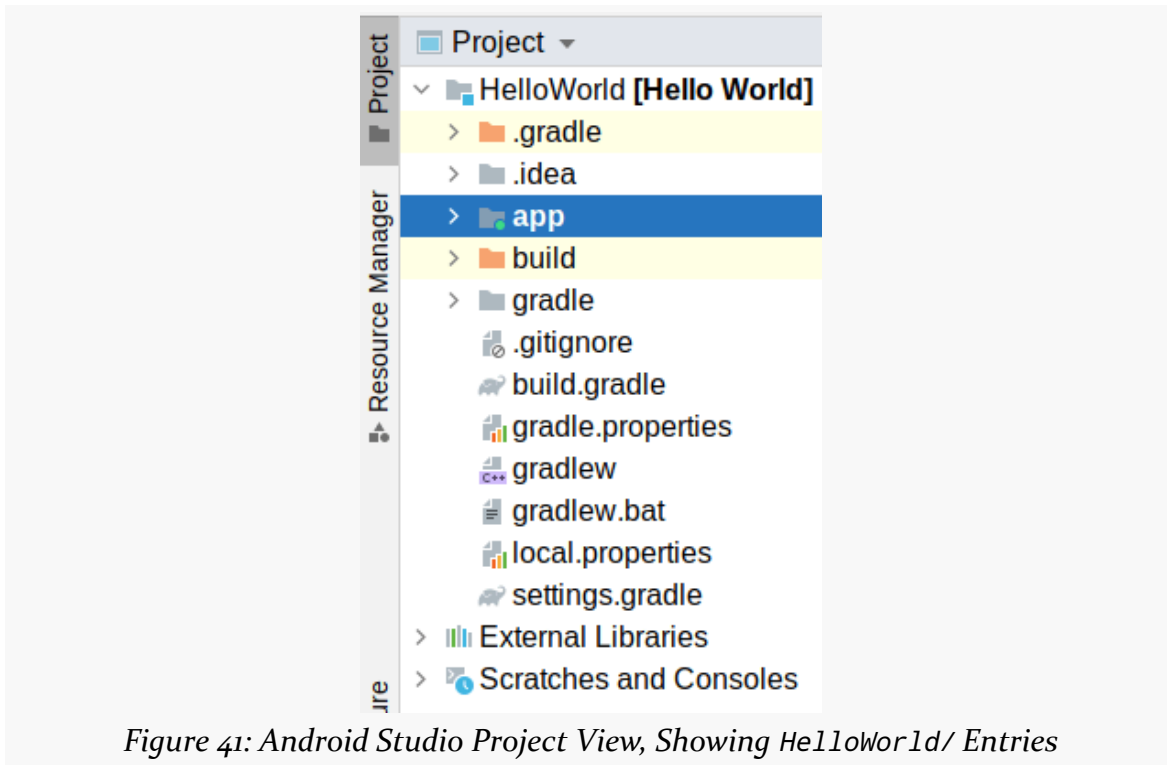


Figure 41: Android Studio Project View, Showing HelloWorld/ Entries

Most of the time, you will be working in the app/ directory. This is called a “module”, and it represents something that you are trying to build:

- An app for an Android device
- An app for some other specialty scenario, such as an app to be deployed to a Wear OS smartwatch
- A library to be used by multiple other modules
- And so on

Your project can have one or several modules; by default, it will just have one, named app/, for building your Android app. In Android Studio Arctic Fox, a module directory is denoted by the small dot or square in the corner of the folder icon and a boldface name.

Some of the files and directories in HelloWorld/ are tied to the Gradle build system,

which we will discuss [later in this book](#):

- `.gradle/`
- `gradle/`
- `build.gradle`
- `gradle.properties`
- `gradlew`
- `gradlew.bat`
- `local.properties`
- `settings.gradle`

The `.idea/` directory, along with the `build/` directory, are generated from the rest of the files in your project. You will not need to do anything with these manually — Android Studio will handle all of that for you.

Android Studio can work with a variety of version control systems, but it has the tightest integration with [Git](#). When you create a project in Android Studio, it will create a `.gitignore` file for you, set up to indicate which files do not need to go into version control. If you are using Git, this file should be a great starting point, though you can modify it as needed (e.g., to ignore other files or directories). If you are not using Git, you can ignore or delete the `.gitignore` file.

The App Module Contents

The `app/` directory contains the files necessary to build your app:

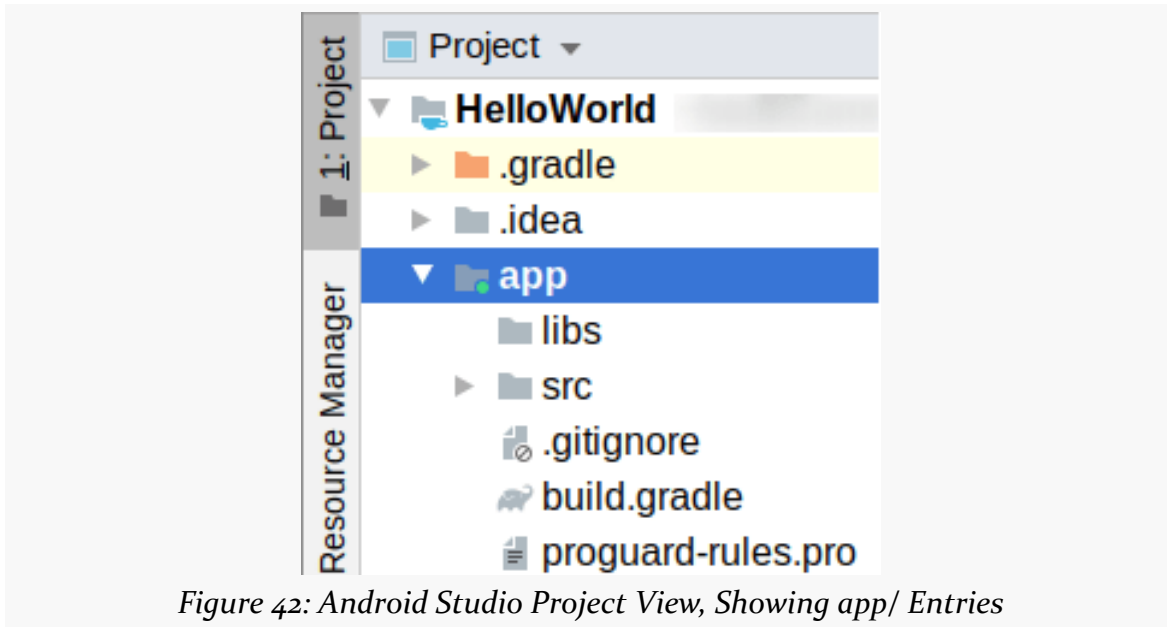


Figure 42: Android Studio Project View, Showing `app/` Entries

Most of the time, you will be working in the `src/` directory, which contains the source code and other files that you will be creating and editing to define your app.

The `build/` directory here is similar to the `build/` directory that is under the `HelloWorld/` root directory. It contains the output of building your app. In this case, we will use this `build/` directory a bit more often, as it contains the actual app itself that we can distribute through the Google Play Store or other app distribution channels. We will examine this `build/` directory a bit more [later in the book](#).

In a typical starter project, the `libs/` directory is empty. It is there to support some old ways of attaching libraries to your module, for code written by others that you wish to use. We will explore what libraries are and what ones this project uses [in an upcoming chapter](#).

The `build.gradle` file, like its counterpart in the `HelloWorld/` root directory, contains Gradle instructions for how to build your app. We will examine this file in detail [later in the book](#).

The `.gitignore` file, like its counterpart in the `HelloWorld/` root directory, identifies

files and directories that can be skipped when putting this project into a Git repository. While you may need to maintain this file if you are using Git, using Git and `.gitignore` is outside the scope of this book.

Last, the `proguard-rules.pro` file is there in support of [ProGuard](#) and related tools. These are used in Android project for two reasons:

1. They help to reduce the size of the apps, by eliminating stuff from libraries that we are not really using
2. They “obfuscate” the code in the apps, to make it slightly more difficult for people to “reverse engineer” the apps and figure out how they work

Until you are ready to think about distributing your app to other users, though, ProGuard and similar tools are not necessary. Hence, we will postpone looking at that stuff until [much later in the book](#).

The Generated Source Sets

Inside the `src/` directory are “source sets”. These identify different directories of source code (and related files) that will be used in different circumstances:

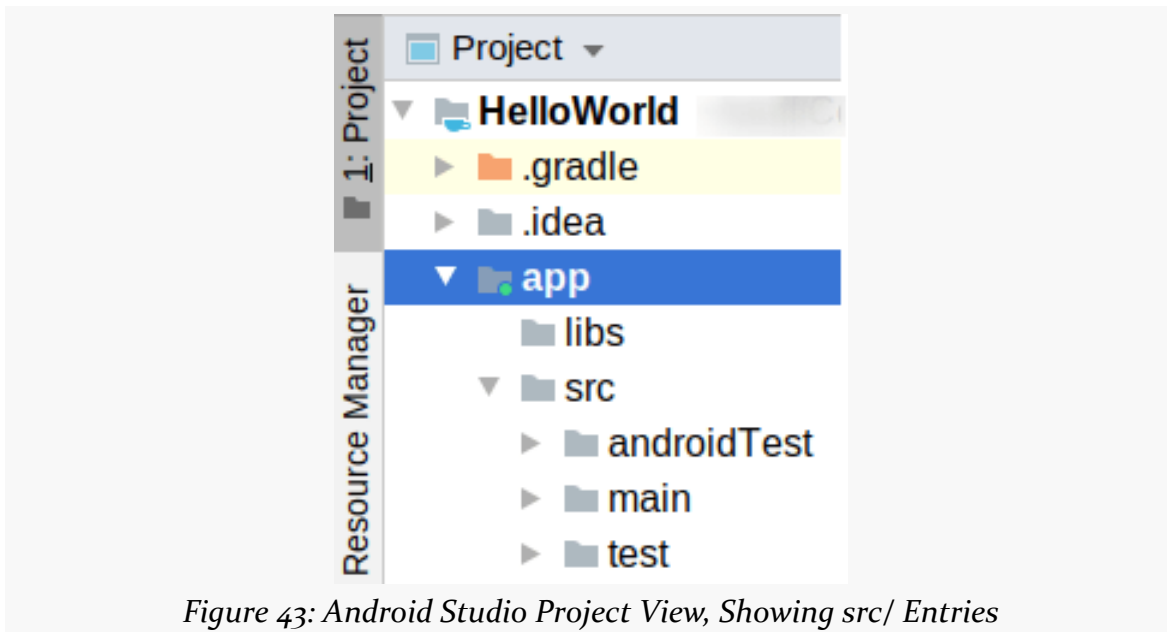


Figure 43: Android Studio Project View, Showing `src/` Entries

There are three source sets that will be created by default for an Android project: `main/`, `androidTest/` and `test/`.

`main/`

The source set that you will spend most of your time in is `main/`. This represents the “main” source code for your app. For most apps, this source set contains all of your code (and related files) that make up the app itself:

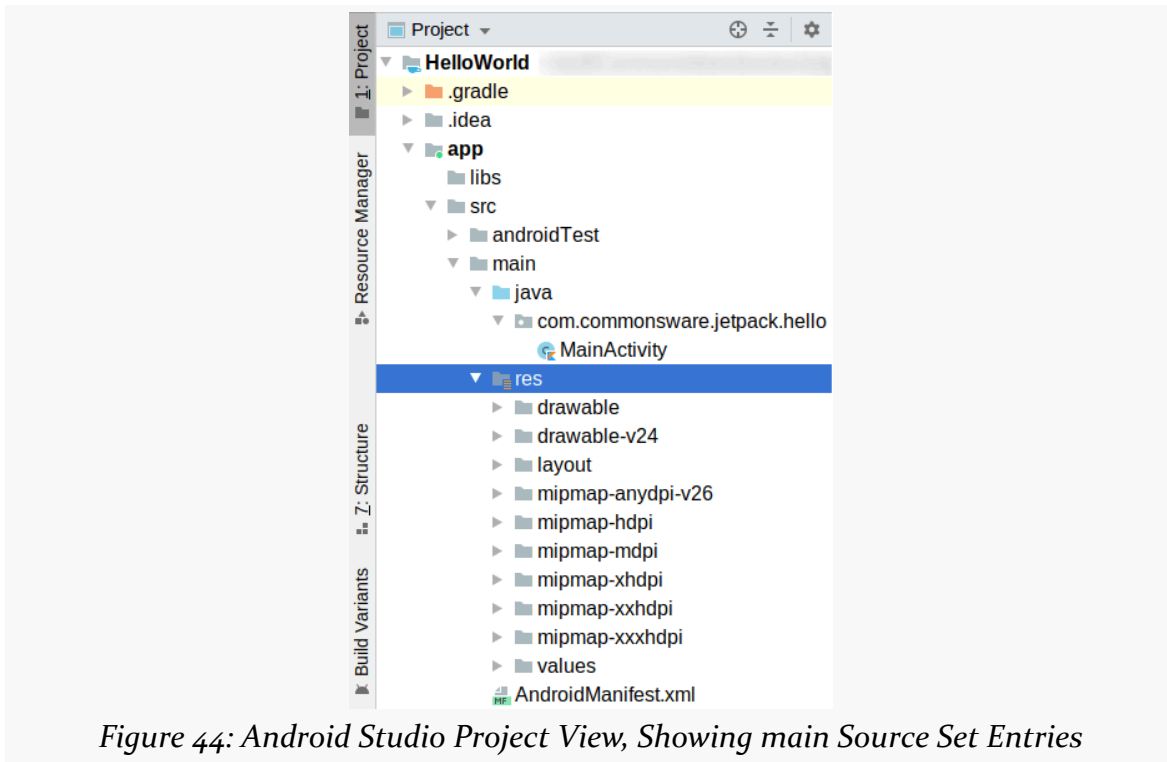


Figure 44: Android Studio Project View, Showing main Source Set Entries

Your Java and Kotlin code will go in a `java/` directory inside of the source set. Here, we see one Kotlin file, named `MainActivity.kt` (where the `.kt` part is left off in the tree).

You will also find:

- Resources, in the `res/` directory, which are files that are not source code but contribute to your app, such as your icons and other images, as we will see [in an upcoming chapter](#)
- `AndroidManifest.xml`, which is the “table of contents” of what is in your app, as we will see [in another upcoming chapter](#)
- Optionally other things (`assets/`, `jni/`, `aidl/`, etc.), though these will not be seen in every Android project

androidTest/

The androidTest/ source set can have its own Java/Kotlin source code, resources, and manifest file. Typically, it only has Java and Kotlin source code:

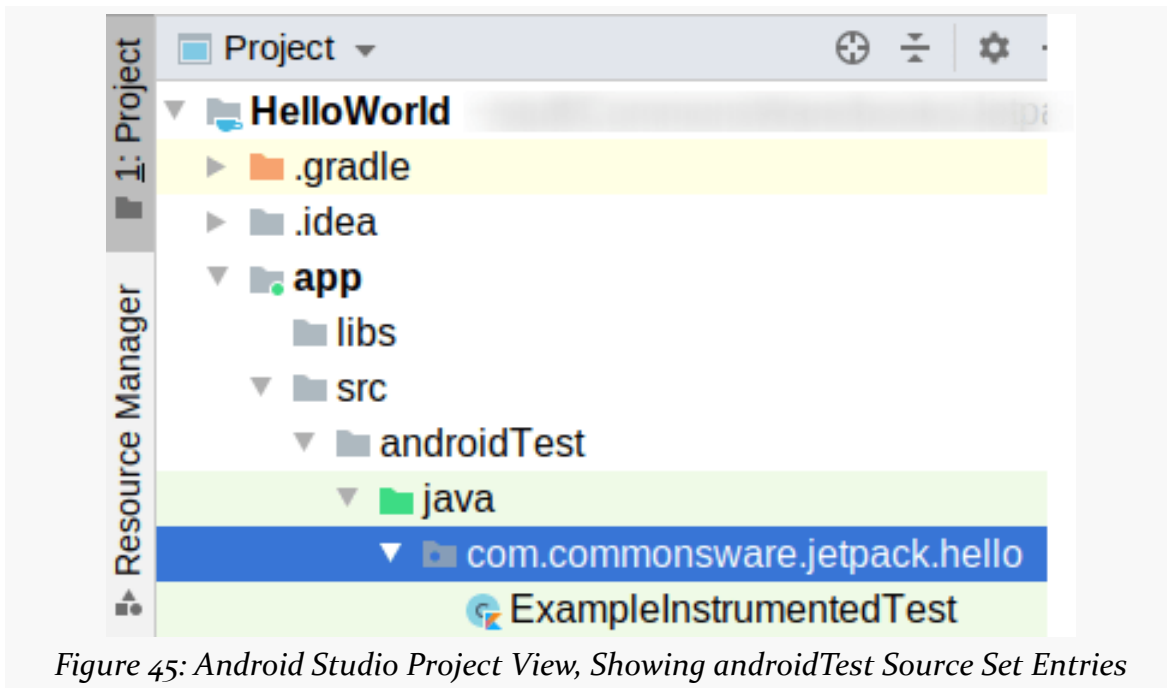
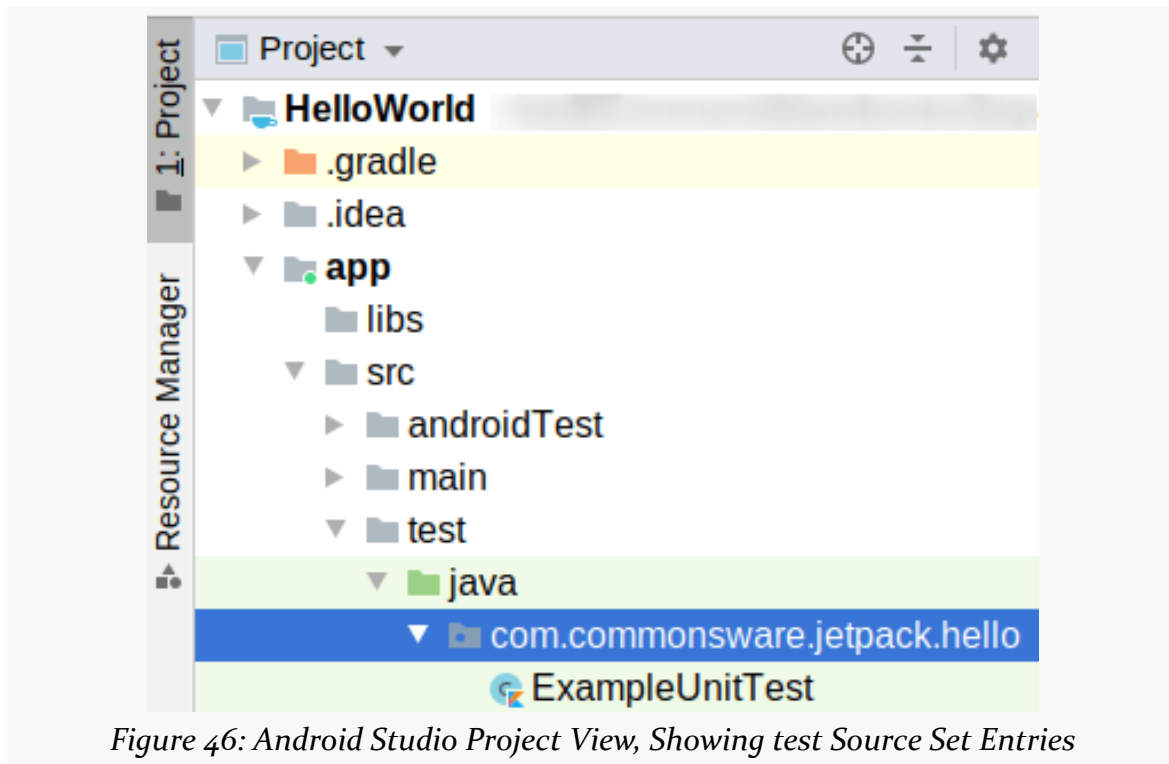


Figure 45: Android Studio Project View, Showing androidTest Source Set Entries

This source set's files will *not* go into your app. Instead, they are for testing your app, to make sure that your app does what it is supposed to do.

test/

There is a very similar source set, named test/, in a typical Android project:



As with androidTest/, test/ can contain its own Java and Kotlin code. And, as with the androidTest/ source set, the test/ code is *not* part of your app, but instead is for testing your app.

The difference between androidTest/ and test/ is in where your tests run:

- androidTest/ tests run inside of an Android device or emulator
- test/ tests run directly on your development machine, in a Java virtual machine

We will explore that distinction in greater detail, along with the test code in our starter project, [later in the book](#).

Language Differences

When you import an Android project into Android Studio, you get whatever source code was in that project. This could be Java, Kotlin, or some combination.

When you create a project from scratch — as we will examine in [an upcoming chapter](#) — you will be able to tell the new-project wizard whether you want it to generate Java or Kotlin files. When you start working on adding new stuff to your project, you can add in new Java or Kotlin files.

The HelloWorld sample project that you imported was created using the Android Studio Arctic Fox new-project wizard, where the author asked for Kotlin files. Therefore, our two test source sets and the main source set contain Kotlin.

If you would prefer, you can download a Java edition of the same project from [the CommonsWare site](#). UnZIPping and importing that project gives you the same thing as the HelloWorld Kotlin project, except that the source code (main/, androidTest/, and test/) is Java, not Kotlin.

Introducing the Activity

Ignoring the test code for a while, our one-and-only source file in our project implements a MainActivity class, either in Java or Kotlin. This class represents an “activity”, one of the core components in an Android app.

The Role of the Activity

The building block of the user interface is the activity. You can think of an activity as being the Android analogue for the window in a desktop application or the page in a classic Web app. It represents a chunk of your user interface and, in many 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 things like a status bar (the strip across the top with the clock, battery icon, etc.) and a navigation bar (the strip across the bottom with buttons for going back, going to the home screen, etc.)

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 multi-window mode on a Chrome OS device. 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).

In a simple app with one activity, such as this sample app, that activity will serve as the entry point for the app itself. The user's home screen will often have an “app drawer” or similar thing with a bunch of icons. While the user thinks of those as “running an app”, in reality those icons pass control to an activity inside of the app, one designated as being something that should appear in a “launcher” or home screen. What makes an activity appear in this app drawer is based upon stuff found in the `AndroidManifest.xml` file, and we will see how that works [later in the book](#).

Examining the Generated Code

When you create a new project via the new-project wizard — as this sample app was — usually you will have that wizard create your first activity for you. The activity will have the same functionality regardless of whether you asked for Java or Kotlin code. Since Java and Kotlin do not have the same syntax, those files will not be identical in code, but they will be identical in functionality.

So, let's see what our sample app's activity looks like, in both languages.

Java

The `HelloWorldJava.zip` version of the sample app was created using Java as the requested programming language. So, our main source set has `MainActivity.java` in a `com.commonware.jetpack.hello` Java package:

```
package com.commonware.jetpack.hello;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

Kotlin

The HelloWorld.zip version of the sample app was created using Kotlin as the requested programming language. So, our main source set has MainActivity.kt in that same Java-style package (com.commonware.jetpack.hello):

```
package com.commonware.jetpack.hello

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Code Commonalities

While Java and Kotlin differ in syntax, both activities are doing the same thing and are using the same things from the Android SDK.

AppCompatActivity

All activities in Android inherit from an android.app.Activity base class. In our case, MainActivity does not inherit directly from that class. Instead, it extends androidx.appcompat.app.AppCompatActivity. That, in turn, inherits from android.app.Activity, so MainActivity has Activity in its inheritance hierarchy.

Technically, you do not need AppCompatActivity — you could inherit from something else, even from Activity itself. However, Google is making it difficult for you to extend from anything else *other* than AppCompatActivity. When you create a new project, it is very likely that you will be given an activity that extends from AppCompatActivity.

The theory is that AppCompatActivity makes it easier for you to develop apps that will behave consistently across many versions of Android, compared to inheriting from Activity or some other subclass of Activity.

We will see more about where AppCompatActivity comes from [a bit later in this chapter](#).

onCreate()

MainActivity has one Java method or Kotlin function: `onCreate()`. This overrides an `onCreate()` method that we inherit. Our job in `onCreate()` is to set up the basic UI that is to be shown by this activity.

In reality, `onCreate()` is just one of a series of “lifecycle methods”, methods or functions that get called as our activity is coming onto the screen, leaving the screen, and so on. We will see more about lifecycles [later in the book](#).

The very first thing that we do in `onCreate()` is chain to the inherited implementation of `onCreate()`, via a call to `super.onCreate()`. This is a very typical pattern for `onCreate()` of an activity, as the activity is not fully initialized until after `super.onCreate()` has been called. So, we try to get that out of the way early, so we are safe to do the rest of our work afterwards.

setContentView()

The other thing that we do in `onCreate()` is call a `setContentView()` method. This says “Hey, Android! The UI that we want to show starts with this!”. We supply something to serve as the foundation for our UI, which we can further tailor if needed. Calling `setContentView()` is not required, but it is a fairly typical approach.

In this case, we pass in a funny-looking value to `setContentView()`: `R.layout.activity_main`. This serves as a reference to a layout resource, named `activity_main`. We will explore resources in [the next chapter](#), including an explanation of [what this R thing is](#).

Line Numbers

You may have noticed that some of the screenshots in this book that show the editing pane show line numbers in the gutter area on the left:

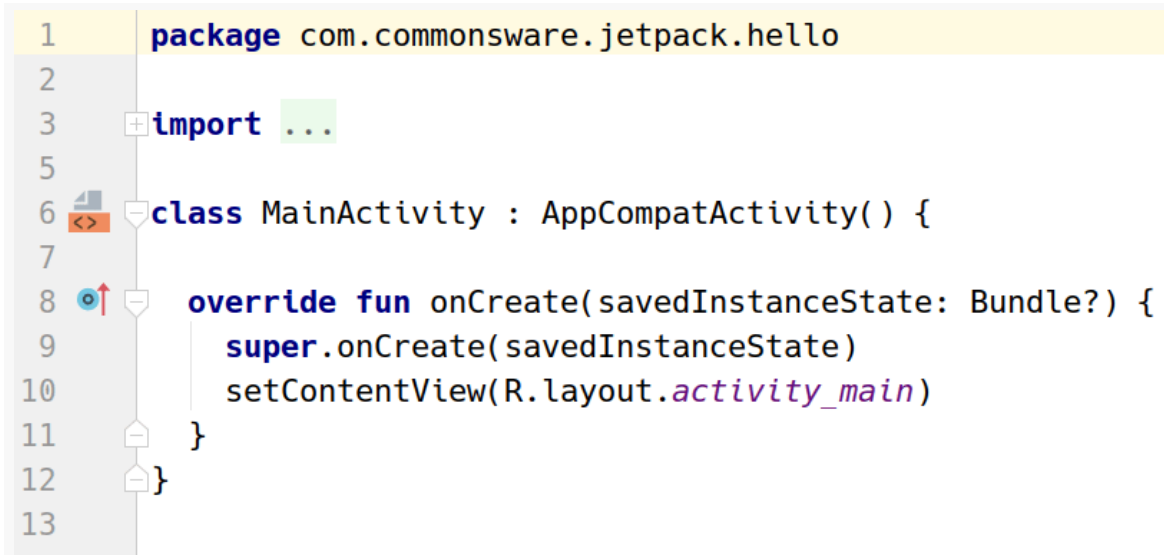


Figure 47: Android Studio Source Editor, Showing Line Numbers

Those will not be enabled by default. If you want to enable them, you have two main options.

EXAMINING YOUR CODE

Per Editor

If you temporarily want to show line numbers, choose “View” > “Active Editor” from the main menu, and toggle on “Show Line Numbers”:

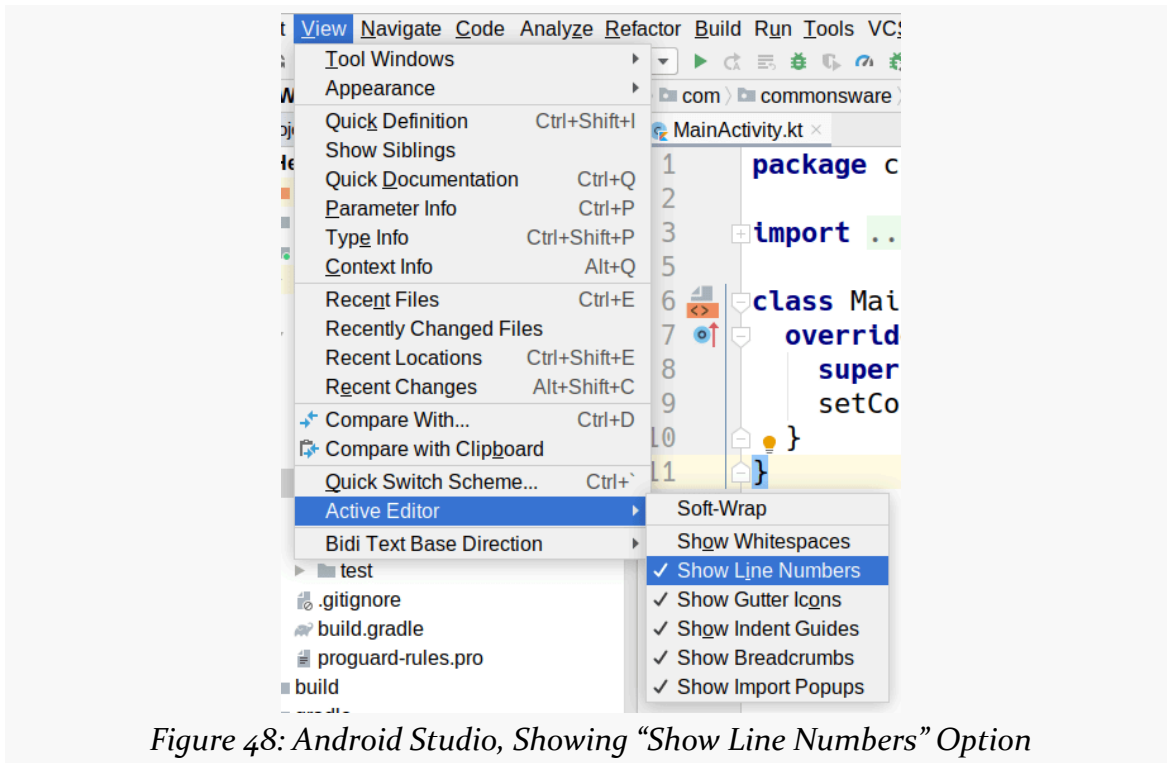


Figure 48: Android Studio, Showing “Show Line Numbers” Option

All the Time

If you wish to have line numbers be toggled on by default, choose “File” > “Settings” from the main menu (or “Android Studio” > “Preferences...” on macOS). Go into “Editor” > “Appearance” and check “Show line numbers”, then click “OK”:

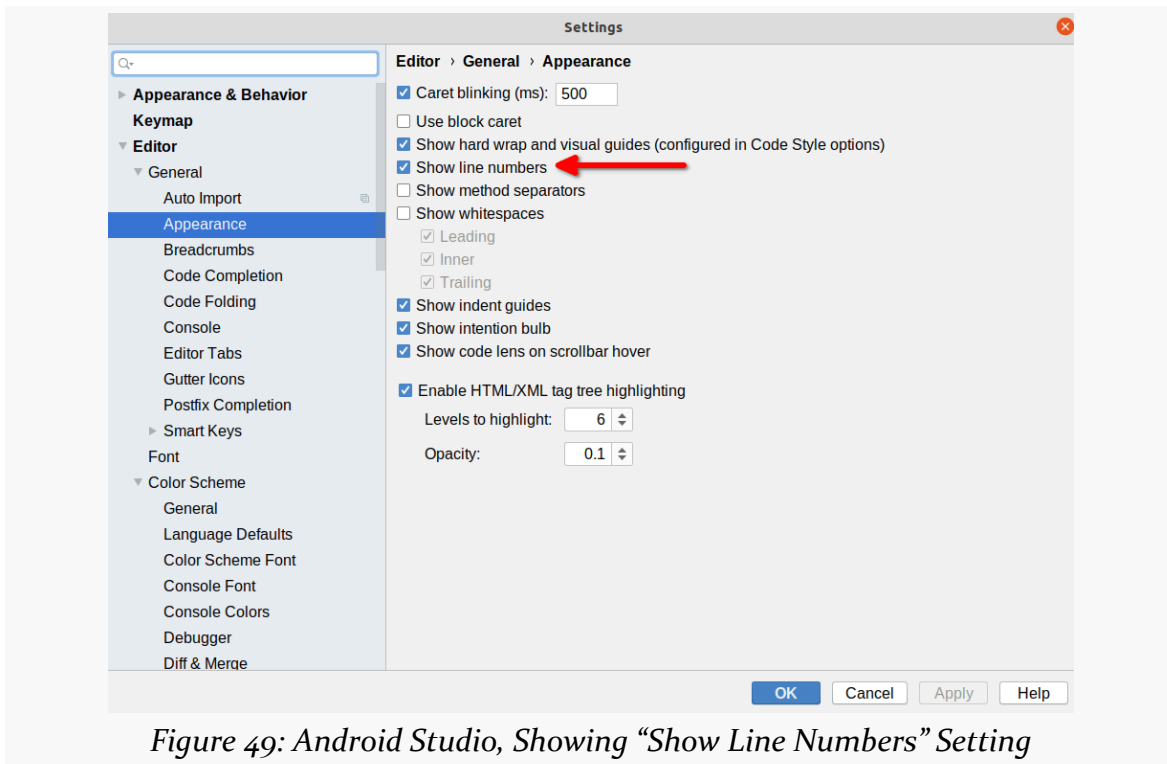


Figure 49: Android Studio, Showing “Show Line Numbers” Setting

We will be exploring other options in this Settings screen throughout the book.

Other Things in the Project Tree

The project tree, starting at the HelloWorld/ root node, contains all of the stuff that makes up the project. However, the project tree itself also contains two other root nodes: “External Libraries” and “Scratches and Consoles”.

External Libraries

The more important of the two, by far, is “External Libraries”, though you will only need to examine this area of the project tree on occasion.

EXAMINING YOUR CODE

MainActivity is fairly small. It can be that small because [“it stands upon the shoulders of giants”](#). In this case, those “giants” are libraries.

Our tiny app pulls in a very long list of libraries:

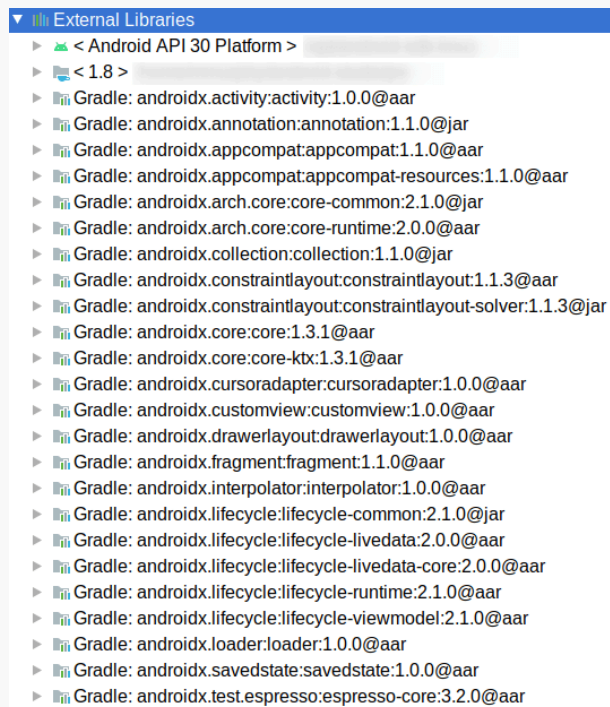


Figure 50: Android Studio External Libraries List (Partial)

This is not a complete list — the list is so long, it cannot fit in a single screenshot.

Most of these libraries come from Google and are part of the Android SDK. Some are from other developers, such as libraries from JetBrains in support of Kotlin.

We will see [in an upcoming chapter](#) where these libraries come from and why they are all being used to build this little app.

Scratches and Consoles

The last item, “Scratches and Consoles”, is almost completely undocumented and seems to be infrequently used. Among other things, via the right-mouse context menu, you can create new “scratch files” here, useful for notes or testing programming language syntax outside the scope of actual project code.

Exploring Your Resources

Resources are static bits of information held outside the Java/Kotlin 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/Kotlin 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/Kotlin 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).

In this chapter, we will examine the resources in our starter project and what their roles are in Android app development. Later chapters will cover more about these resources and describe other types of resources that your project can have.

What You See in res/

If you look at the `app/src/main/res/` directory of your project, you will see a fairly long list of subdirectories:



Figure 51: Resource Directories in Android Studio

Resources are placed into directory based in part on the resource type. That forms the base name of the directory, such as `drawable` and `layout`. Some directories contain a suffix after this, such as the `-v24` part of `drawable-v24`. That suffix indicates a particular resource set, which we will examine more [shortly](#).

But first, we need to talk about API levels.

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

EXPLORING YOUR RESOURCES

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 techniques for creating 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.

We started with API Level 1 and Android 1.0... but that was a long time ago. Nowadays, the focus tends to be on newer versions of Android and corresponding higher API levels. Here, though, Android gets a bit complicated, as there are a *lot* of different versions of Android being used today.

Google used to publish up-to-date version information on its [dashboards page](#), but they abandoned that a while ago. Instead, they rely on you [creating a new project](#) and looking at a version distribution chart available from the new project wizard.

EXPLORING YOUR RESOURCES

Or, you could just look at the copy of that chart shown here:

ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
4.0 Ice Cream Sandwich	15	
4.1 Jelly Bean	16	99.8%
4.2 Jelly Bean	17	99.2%
4.3 Jelly Bean	18	98.4%
4.4 KitKat	19	98.1%
5.0 Lollipop	21	94.1%
5.1 Lollipop	22	92.3%
6.0 Marshmallow	23	84.9%
7.0 Nougat	24	73.7%
7.1 Nougat	25	66.2%
8.0 Oreo	26	60.8%
8.1 Oreo	27	53.5%
9.0 Pie	28	39.5%
10. Android 10	29	8.2%

Figure 52: Android OS Distribution Chart, from Android Studio 4.2.1

The chart shows the various Android versions, their API level numbers, and the “cumulative distribution”. The cumulative distribution shows you what percentage of the Android device ecosystem you can reach if your `minSdkVersion` is set to that particular API level. Their numbers are based on devices using the Play Store and therefore will miss many devices that are based on other distribution channels.

This book focuses on Android 5.0 (API Level 21) and higher. There are ways to support older devices than that, but supporting older than Android 4.4 (API Level 19) gets complicated, so this book skips that to help keep the explanations simple.

At the time that this chapter was last updated, the latest production version of Android was 12 (API Level 31). This is not shown in the chart, because [Google appears to have forgotten to update the chart in over a year](#).

Beyond the latest production version, from time to time we are given “developer previews” of an upcoming version of Android. These are not good choices for new Android developers to worry about, but experienced developers may be interested in testing on pre-release Android versions and trying to use upcoming features. For

example, in February 2021, Google announced “Android S” and released the first developer preview of what became Android 12.

Decoding Resource Directory Names

Our app has a bunch of resource directories. Some have simple names, like `drawable/` and `layout/`. Others have suffixes, like `drawable-v24/` and `mipmap-hdpi/`.

The initial segment of the directory name — or the whole name for those that lack suffixes — usually indicates the type of the resource. There are many resource types in Android, and we will explore a few of them in this chapter. The exception is the `values/` directory, which can contain a variety of smaller resource types, not just one “values” type.

The suffixes represent “resource sets”, and they indicate that this directory contains resources of a particular type that should only be used in certain scenarios. We call these scenarios “configurations”; the suffixes indicate what configurations those resources are used for.

For example:

- `drawable/` has resources that are good for any configuration, but `drawable-v24/` has resources that are only going to be used on API Level 24 and higher devices (i.e., Android 7.0 and higher)
- `mipmap-anydpi-v26/` has resources that are good for any screen density, but they will only be used on API Level 26 and higher devices
- `mipmap-mdpi/` has resources that are designed around “medium density” screens, where the density is around 160dpi (dpi = dots per inch)

We will explore these rules more [later in the book](#), as they get fairly complex fairly quickly.

Our Initial Resource Types

Our starter app contains six types of resources, though two of them (drawables and mipmaps) are pretty much the same thing.

Layouts

The resource type that will consume most of your time is the layout resource. This

describes a chunk of our app's user interface. That chunk could be:

- A screen
- A row in a list
- A cell in a grid
- A reusable piece that you want to apply to several different screens
- And so on

Layout resources are XML files that either you create by hand or create through the use of drag-and-drop GUI builders built into Android Studio. We will be spending quite a bit of time covering layout resources throughout this book, starting with [a chapter on widgets](#), our smallest pieces of a layout resource.

The starter project has a fairly simple layout... though it could be even simpler:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

As we will see, XML elements generally will map to widgets (things that users touch) and containers (things that organize widgets and other containers). So, here, we have a `ConstraintLayout` container that wraps around a single `TextView` widget.

In [the chapter on widgets](#), we will explore this XML structure in detail, plus show you how you can set up this XML through the drag-and-drop GUI builder.

Drawables and Mipmaps

All apps have some amount of artwork, mostly in the form of icons. For example, most apps have an icon that will appear in the home screen or launcher app, that allows the user to bring up the app's UI. Apps might have other icons in that UI, to appear on buttons or other tappable things. Some apps may use a “splash screen” as an introductory bit of UI, and so they have some large graphic that they want to use for that screen. And there are many other uses of artwork within an Android app, of relevance to some apps but perhaps not to others.

Sometimes, these graphics are downloaded from the Internet as part of running the app. Most of the rest are packaged with the app itself: the graphic designer creates the artwork and the developer arranges to use it in the app in the appropriate place.

Most of these pre-packaged bits of artwork are in the form of drawable and mipmap resources.

Many of these are bitmap images: PNG, JPEG, etc. They can also be:

- Vector art, imported from SVG files that your graphic designer might prepare in tools like Adobe Illustrator
- Specialized XML files, usually with rules for how to combine two or more other resources together

There are really boring technical distinctions between drawables and mipmaps, and tedious historical explanations for why we have two different resource types for the same stuff. For the purposes of this book — and so you do not fall asleep while reading it — you do not need to worry about all of that. The rules for the vast majority of Android developers are fairly simple:

- Your home screen launcher icon is a mipmap
- Everything else is a drawable

We will start exploring these resources more in an upcoming chapter where [we change your launcher icon](#).

Strings

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

EXPLORING YOUR RESOURCES

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.

Strings are one of the “values” resource types. So, in the `values/` directory, we can have one or several files that contain string resources. Typically, you have just one such file, named `strings.xml`.

The starter app’s `strings.xml` file contains... not very much:

```
<resources>
  <string name="app_name">Hello World</string>
</resources>
```

All files in the `values/` directory will be XML files with a root `<resources>` element. What appears inside that root element defines the actual resources contained in that file.

Inside `strings.xml`, the `<resources>` element contains just one child element: a `<string>`, defining a single string resource. Each string resource has a name, which is how we will refer to that string from elsewhere in the app. And, each string resource has a value, consisting of the text between the `<string>` and `</string>` tags. Here, we define `app_name` to be “HelloWorld”.

The starter app does not have translations of this resource, but it could. For example, it could contain a `res/values-es/` directory, containing strings to be used for devices whose locale is set to Spanish. In there, `app_name` might be defined as “Hola Mundo”. On the fly, Android will choose the right translation to use, based on the translations that you provide and the locale of the device.

We will be working with a bunch of string resources in this book, and we will explore the issues of translations a bit more [in a later chapter](#).

Colors

Another type of “values” resource is the color resource. As you might expect, it provides a symbolic name for colors. This allows us to give names that have semantic meaning (e.g., “the standard accent color”) and use those names in our code. It also then gives us one place to define what the actual color is for that name, so if we need to change the color, we can change it in one place.

Color resources are defined by `<color>` elements in a “values” resource file.

EXPLORING YOUR RESOURCES

Convention says that your colors go into a `colors.xml` resource file, and that is what the starter app has:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="purple_200">#FFBB86FC</color>
  <color name="purple_500">#FF6200EE</color>
  <color name="purple_700">#FF3700B3</color>
  <color name="teal_200">#FF03DAC5</color>
  <color name="teal_700">#FF018786</color>
  <color name="black">#FF000000</color>
  <color name="white">#FFFFFFFF</color>
</resources>
```

As with our `app_name` string resource, just having these colors does not cause anything to *use* those colors. That requires additional code or additional resources, ones that happen to reference these resources.

Styles and Themes

A place where color resources are often used is in style resources. Style resources are reminiscent of CSS stylesheets in Web development. Styles allow you to give a name to a collection of UI properties, then apply those properties to various scenarios.

One such scenario is where a style is used as a “theme”. This provides the defaults for UI properties for an entire activity, or perhaps even the entire app. The sample project defines one such theme, `AppTheme`, in its `themes.xml` file:

```
<resources xmlns:tools="http://schemas.android.com/tools">
  <!-- Base application theme. -->
  <style name="Theme.HelloWorld" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
    <!-- Primary brand color. -->
    <item name="colorPrimary">@color/purple_500</item>
    <item name="colorPrimaryVariant">@color/purple_700</item>
    <item name="colorOnPrimary">@color/white</item>
    <!-- Secondary brand color. -->
    <item name="colorSecondary">@color/teal_200</item>
    <item name="colorSecondaryVariant">@color/teal_700</item>
    <item name="colorOnSecondary">@color/black</item>
    <!-- Status bar color. -->
    <item name="android:statusBarColor" tools:targetApi="1"
        ?attr/colorPrimaryVariant
    </item>
    <!-- Customize your theme here. -->
  </style>
</resources>
```

The parent attribute on the `<style>` indicates that we are inheriting existing UI

EXPLORING YOUR RESOURCES

property definitions from something called

`Theme.MaterialComponents.DayNight.DarkActionBar`. That name has lots of pieces:

- Theme indicates that we are starting from the base system theme
- MaterialComponents indicates that this theme comes from a library referred to as [the Material Components for Android](#)
- DayNight indicates that the general look is dark text on a light background during the daytime, and light text on a dark background at night
- DarkActionBar says that the “app bar” — the toolbar structure towards the top of the screen of most Android activities, formerly called an “action bar” — should have white icons on a dark background

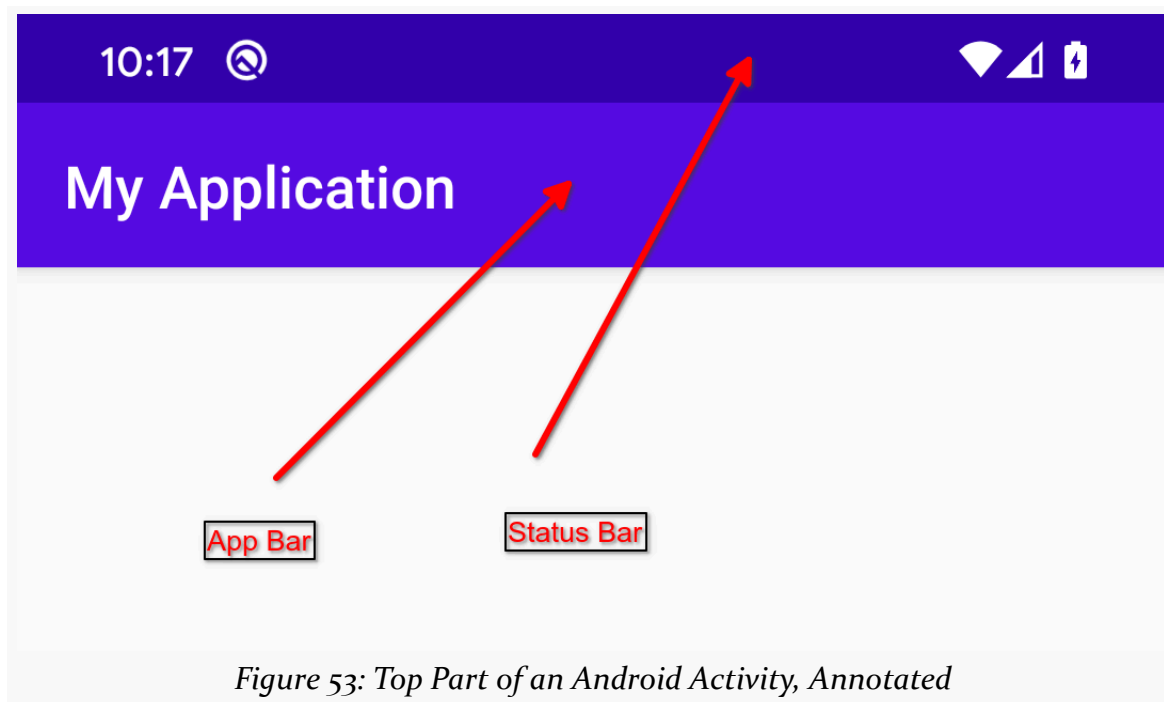


Figure 53: Top Part of an Android Activity, Annotated

`AppTheme` inherits from `Theme.MaterialComponents.DayNight.DarkActionBar`, so we get lots of stuff “for free” as a result. We then override additional UI properties as we see fit, such as `colorPrimary`, which the Material Components will use for the app bar background, the foreground text in the main UI area, and a few other roles.

The `AppTheme` style refers to the color resources that were defined in the `colors.xml` file. In resources, when you need to refer to another resource, you do so using the syntax `@type/name`, where `type` is the type of the resource (color, string, drawable, mipmap, etc.), and `name` is the name of the resource. For “values” resources, like our colors, the name comes from the `name` attribute of the element that defines the

resource. For all other types of resources, the name comes from the filename of the resource file, without the file extension. So, here, `@color/colorPrimary` refers to the `colorPrimary` color resource.

Note that the resource references do *not* include any of those suffixes on the directory names that we use for resource sets. If you look in the various directories for mipmap resources, you will see that we have six different variations of an `ic_launcher` mipmap:

- `mipmap-anydpi-v26/ic_launcher.xml`
- `mipmap-hdpi/ic_launcher.png`
- `mipmap-mdpi/ic_launcher.png`
- `mipmap-xhdpi/ic_launcher.png`
- `mipmap-xxhdpi/ic_launcher.png`
- `mipmap-xxxhdpi/ic_launcher.png`

However, when things like [our manifest](#) refer to these, it is always as `@mipmap/ic_launcher`. Android will decide, on the fly, which of these six definitions to use, based on the rules encoded in those directory names and the configuration of the device at the time we are trying to use the resource. We will get much more into all of that complexity [later in the book](#).

About That R Thingy

When we were looking at the source code to `MainActivity`, we saw this line:

```
setContentView(R.layout.activity_main)
```

`setContentView()` tells the activity “this is the UI to display”. The `R.layout.activity_main` value is a reference to our `activity_main.xml` layout resource.

Just as we refer to our app’s resources from other resources using `@type/name` syntax, we refer to our app’s resources from Java and Kotlin using `R.type.name` syntax. The same rules apply:

- The type is the type of the resource (e.g., `layout`), not counting any suffixes that might be on the directory name
- The name is the name attribute of a “values” resource or the filename of other types of resources, excluding the file extension

Occasionally, you will try to refer to an R value and the IDE will say that it cannot find that value. We will explore this problem more in [a bit later in the book](#).

The Resource Manager

Android Studio also has a “Resource Manager” tool. This is designed to help you navigate some key types of resources more easily, such as:

- Drawables and mipmaps
- Colors
- Layouts
- Strings

This tool is accessible via a “Resource Manager” button, docked by default on the left edge of the Android Studio window.

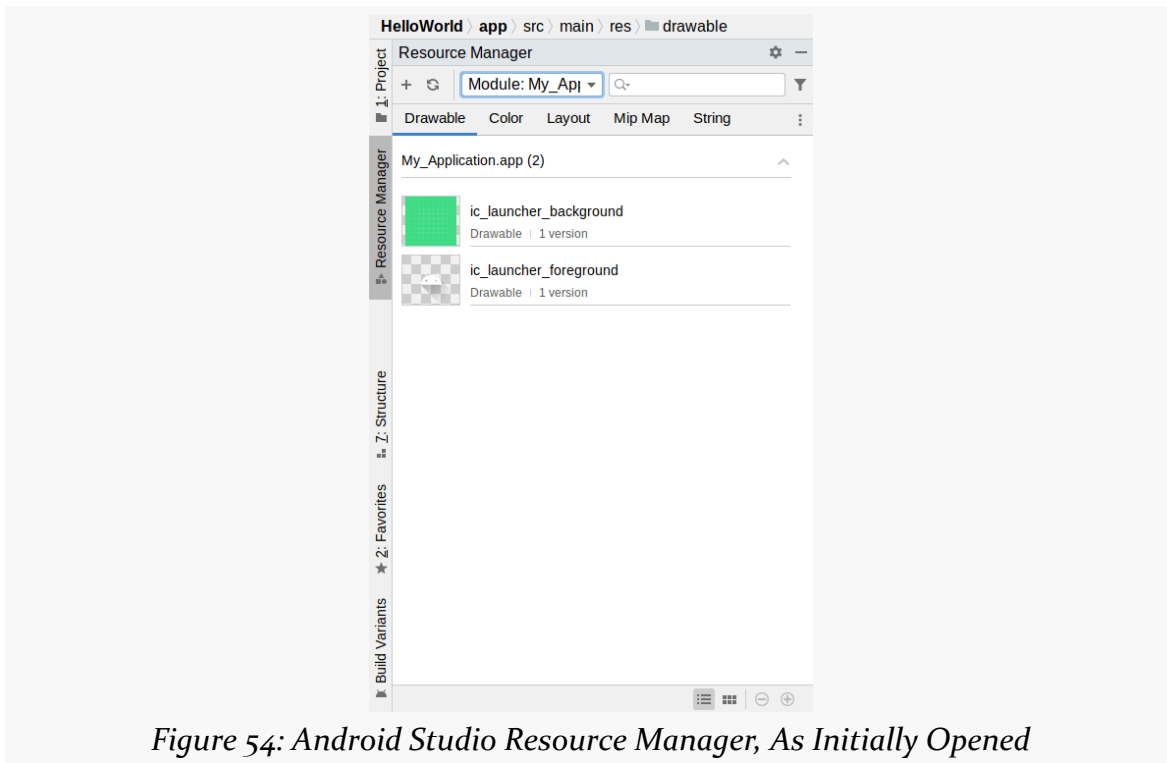
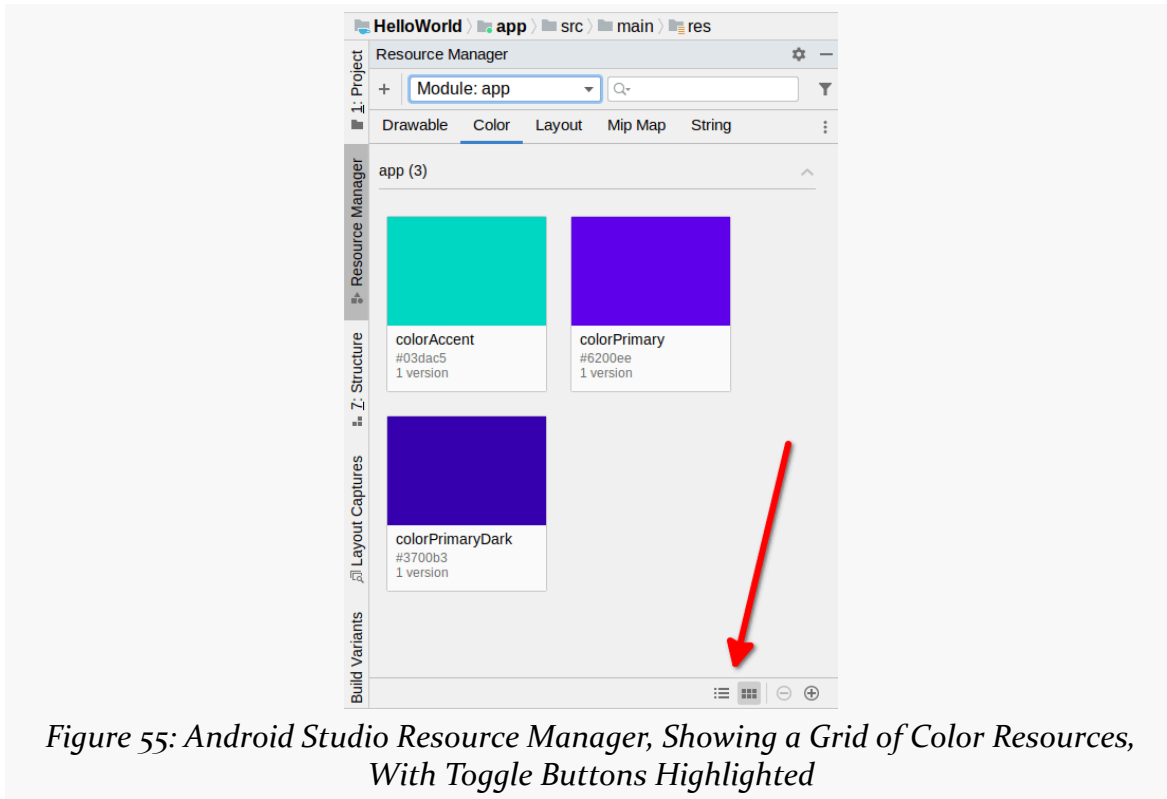


Figure 54: Android Studio Resource Manager, As Initially Opened

There are a set of tabs towards the top that allow you to toggle between the three sets of resources that this tool supports. By default, each tab’s contents is a list, but there are buttons towards the bottom of the tool to toggle between list and grid

EXPLORING YOUR RESOURCES

modes:



Double-clicking on a resource opens up an editor window for it, while right-clicking over a resource provides options for copying it, renaming it, etc.

The + icon in the toolbar towards the top lets you add new drawable resources, using the Image Asset Wizard or Vector Asset Wizard, both of which we will see in upcoming chapters.

For larger projects, this can be useful to help you find resources. In particular, the Drawable tab can be very handy for identifying if you already have a piece of artwork in your app or whether you need to add it.

Inspecting Your Manifest

A key part of 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, such as your activities. 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 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 require a few minor modifications, but otherwise it will be fine. Some apps will have a manifest that has 1,000+ lines. Your production Android applications probably will fall somewhere in the middle.

The Root Element

Here is the `AndroidManifest.xml` file from the starter project:

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

    <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/Theme.HelloWorld">
```



```
<activity
  android:name=".MainActivity"
  android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

The root `<manifest>` element usually does not contain too much. It will have one or more XML namespace declarations. Here, we have just one, defining the `android` namespace, which is used for most of the attributes that you will find in the manifest file. We will see other manifests later on that have other namespace declarations (e.g., `tools`), but usually there are not too many of them.

The key attribute in the `<manifest>` element is `package`. This indicates where the build tools will generate some Java code for use by your app. We will explore that generated code [later in the book](#).

The Application Element

There can be many child elements of the root `<manifest>` element. Over the course of this book, we will see ones like `<uses-permission>` that appear in these manifests.

However, the most important child element by far is `<application>`. This describes the app that is using this manifest.

In a significant Android app, most of what goes in the manifest consists of child elements of `<application>`, such as [the `<activity>` element](#). Beyond that, the `<application>` element:

- Provides defaults for behavior of those activities, such as what theme is used to specify colors and such (`android:theme`)
- Provides details about the app that get used by other apps (e.g., Settings), such as the app's display name (`android:label`) and icon (`android:icon` and, sometimes, `android:roundIcon`)
- Configures overall app behavior, such as whether it handles right-to-left

- languages (a.k.a., RTL), such as Arabic and Hebrew (`android:supportsRtl`)
- Configures certain aspects of how the app integrates with the rest of the operating system, such as whether it wishes to participate in device-wide backups (`android:allowBackup`)

We will explore many of these attributes as we proceed in this book.

The Activity Element (And Its Children)

The children of `<application>` mostly represent the “table of contents” for the app.

Android has four major types of “components”:

- Activities, representing the UI
- Services, representing background processing that is decoupled from the UI
- Content providers, which expose databases or data streams to other apps or the operating system
- Broadcast receivers, which supply the “subscriber” side of a publish/subscribe messaging system used by apps and the operating system to communicate

Most of these will be registered in the manifest via corresponding child elements of `<application>`:

Component	Element
Activity	<code><activity></code>
Service	<code><service></code>
Content Provider	<code><provider></code>
Broadcast Receiver	<code><receiver></code>

Your app may have several of one type, such as having several activities. Your app may have none of a particular type, such as having no broadcast receivers registered in the manifest.

Our starter app has a single `<activity>` element, and nothing more:

```
<activity
  android:name=".MainActivity"
  android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />

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

`<activity>` elements have an `android:name` attribute. This will identify the Java or Kotlin class that contains the implementation of the activity. The `android:name` attribute, in this case, has a bare Java class name prefixed with a single dot (`.MainActivity`). Sometimes, you will see `android:name` with a fully-qualified class name (e.g., `com.commonsware.helloworld.MainActivity`). Sometimes, you will see just a bare Java class name (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.

Sometimes, an `<activity>` element will have an `<intent-filter>` child element describing under what conditions this activity will be displayed. Most apps will have at least one `<activity>` element that sets up your activity to appear in the launcher, so users can choose to run it. That is what this `<intent-filter>` element does, though the details of how that works are beyond the scope of this particular book. Suffice it to say that whenever you see an `<activity>` element with this particular `<intent-filter>` (an `<action>` of `android.intent.action.MAIN` and a `<category>` of `android.intent.category.LAUNCHER`), you know that this activity should appear in the launcher for the user to be able to start.

The other component elements — `<service>`, `<provider>`, `<receiver>` — will have similar characteristics:

- They all will have an `android:name` attribute, identifying the code that serves as the implementation for that component
- They might have an `<intent-filter>`
- They might have other attributes as well (e.g., `android:permission`)

Reviewing Your Gradle Scripts

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...)

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, as used by default in Android Studio Arctic Fox, 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. Kotlin is one of particular importance for Android developers. [Groovy](#) is another.

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

Groovy also resembles Kotlin in some ways:

- You can have free-standing statements outside of a class
- You can use string interpolation (e.g., "Hello, \$name" to dynamically insert a name value into the string)
- You can skip types on variable declarations (e.g., `def foo = 1;`, akin to `var foo = 1` in Kotlin)

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.

Hey, I Thought I Read That Gradle Used Kotlin Scripts?

There is an option, starting with Android Studio 4.0, to use Kotlin scripts for defining your Gradle builds, instead of Groovy scripts. If you see a project with `build.gradle.kts` files instead of `build.gradle` files, that project is using Kotlin Gradle scripts instead of Groovy ones.

This is likely to prove to be the long-term direction for Android. However, this book is going to focus on Groovy scripts, for a few reasons:

- The new-project wizard still generates Groovy scripts
- The *vast* majority of existing projects — such as the one you might start helping on — will be using Groovy scripts

- In the end, both scripts wind up doing the same work, with the sole difference being some syntax variances between Groovy and Kotlin

Obtaining Gradle

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 outside of an IDE, such as using a continuous integration (CI) server, like Jenkins or Circle CI.

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

Direct Installation

What some 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.

OS Packages

You may be able to obtain Gradle via [a package manager](#) for your particular operating system

The gradlew Wrapper

A brand new Android Studio project — and many of those that you will find in places like GitHub — will have 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`)

REVIEWING YOUR GRADLE SCRIPTS

- 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:

```
#Sat Sep 11 19:07:11 EDT 2021
distributionBase=GRADLE_USER_HOME
distributionUrl=https\://services.gradle.org/distributions/gradle-7.0.2-bin.zip
distributionPath=wrapper/dists
zipStorePath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
```

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 something that you download from GitHub — 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, as it may have been modified by some malware author.

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.

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. The starter project has the single app module, and many projects only need one module.

If you downloaded the Kotlin edition of the starter project, your top-level `build.gradle` looks like this:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
buildscript {
    repositories {
        google()
        mavenCentral()
    }
    dependencies {
        classpath "com.android.tools.build:gradle:7.0.2"
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.30"

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

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

If instead you have the Java-based starter project, you will have a very similar `build.gradle` file, one without the Kotlin references:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
buildscript {
    repositories {
        google()
        mavenCentral()
    }
    dependencies {
```

REVIEWING YOUR GRADLE SCRIPTS

```
classpath "com.android.tools.build:gradle:7.0.2"

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

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

In either case, the file contains three things (besides an opening comment):

- a `buildscript` closure
- an `allprojects` closure
- a `clean` task

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

There are three main closures in the project-level `build.gradle` file.

`buildscript`

The `buildscript` closure in Gradle is where you list sources of plugins that you want to use in the project. 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 libraries that contain Gradle plugins. In the Kotlin edition of the `build.gradle` file, there are two such dependencies:

- `com.android.tools.build:gradle`, which is where the Android Plugin for Gradle comes from, which teaches Gradle how to build Android apps
- `org.jetbrains.kotlin:kotlin-gradle-plugin`, which teaches Gradle how to compile Kotlin source code

(the Java edition of the project will lack the Kotlin plugin)

The identifiers of the libraries (e.g., `com.android.tools.build:gradle`) are followed

REVIEWING YOUR GRADLE SCRIPTS

by a version number, indicating what particular version of those libraries should be used. From time to time, Android Studio will ask you to update those versions, just as it will ask on occasion for you to upgrade the version of Gradle specified in `gradle-wrapper.properties`. Google maintains a page [listing the Gradle versions supported by each Android Gradle Plugin version](#)

`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. We will talk a bit more about libraries [later in this chapter](#).

`clean`

Like many build systems, Gradle is based around tasks. Plugins and your own Gradle files teach Gradle about various tasks that it should be able to perform when requested. The `clean()` task in the top-level `build.gradle` file is one such task. As written, this task is almost useless, and it is unclear why Google includes it, other than perhaps to point out that you are able to define custom tasks.

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.

The Kotlin project’s edition of `app/build.gradle` includes a number of Kotlin references:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
}  
  
android {  
    compileSdk 31  
  
    defaultConfig {  
        applicationId "com.commonware.jetpack.hello"  
        minSdk 21  
        targetSdk 31  
        versionCode 1  
        versionName "1.0"  
    }  
}
```

REVIEWING YOUR GRADLE SCRIPTS

```
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
kotlinOptions {
    jvmTarget = '1.8'
}
}

dependencies {

    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'com.google.android.material:material:1.4.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

...while the Java edition does not:

```
plugins {
    id 'com.android.application'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.jetpack.hello"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

```
}  
dependencies {  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'com.google.android.material:material:1.4.0'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
    testImplementation 'junit:junit:4.+'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
}
```

The android closure contains all of the Android-specific configuration information. The Android plugin will use this closure, where the plugin itself comes from the id 'com.android.application' line at the top, coupled with the classpath line from the project-level build.gradle file. We will explore some of the specific values defined in this closure [in the next section](#).

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 talk more about libraries [later in the chapter](#).

Requesting Plugins

The first lines in app/build.gradle usually request various plugins. The lines that you added to the top-level build.gradle file specify *sources* of plugins, but those libraries can have many different plugins, and you only need some of them. Plus, if your project grows and you have more modules than just app, you might need a different mix of plugins per module.

As a result, each module's build.gradle file starts off with a plugin closure to indicate what plugins are needed:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
}
```

Both the Java and the Kotlin editions of the starter project will request the com.android.application plugin. This teaches Gradle how to build Android apps. There are other options here, such as com.android.library to teach Gradle how to build an Android library, but nearly every project will have at least one module using

`com.android.application`.

Kotlin-based projects will also request the `kotlin-android` plugin. This teaches Gradle how to compile Kotlin code, particularly in the context of building an Android application.

Android Plugin for Gradle Configuration

One of the most important areas for configuration in `app/build.gradle` is inside the `android` closure. That configures the Android Plugin for Gradle, teaching it the details of how you want your app to be assembled from its source code, resources, etc.

In theory, this closure can get *very* complex. In practice, most apps will configure just a few things, as the starter app does:

```
android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonware.jetpack.hello"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = '1.8'
    }
}
```

Package Name and Application ID

In [the previous chapter](#), we saw that there is a `package` attribute on the root `<manifest>` element of a manifest, and that indicates where some code-generated classes will go.

The package value also serves as the default value for your application ID. However, you can override it, as the starter app does, via an `applicationId` statement in the `defaultConfig` closure inside that `android` closure.

The application ID is a unique identifier for our app, 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)

Convention says that the application ID starts with a reversed edition of some domain name that you control, to reduce the likelihood of an accidental collision with the application ID of some other developer.

`compileSdkVersion`, `minSdkVersion`, **and** `targetSdkVersion`

Back in the chapter introducing resources, we discussed [the concept of API levels](#). API levels are integers, with higher numbers indicating newer versions of Android. A new API level is created for:

- Every major version of Android (e.g., Android 8.0 is API Level 26)
- Every minor version of Android (e.g., Android 8.1 is API Level 27)
- Some patch versions of Android (e.g., Android 4.0 was API Level 14, but Android 4.0.3 was API Level 15)

We use those API levels in three key places in the module's `build.gradle` file: `compileSdkVersion`, `minSdkVersion`, and `targetSdkVersion`.

`compileSdkVersion` indicates what version of Android do we want to compile against. Classes, methods, and other symbols that existed in Android at that time (or from before) will be available to us at compile time, but newer things will not. Usually, therefore, we set the `compileSdkVersion` to be a fairly modern API level, such as the latest production version of Android.

`minSdkVersion` indicates what is the oldest version of Android you are willing to support. So, if you are only supporting your app on Android 5.0 and newer versions of Android, you would set your `minSdkVersion` to be 21. Older devices will be incapable of installing your app, and your app will not appear in the Play Store app for devices running an older version of Android.

`targetSdkVersion` 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. However, from a practical standpoint, nowadays the `targetSdkVersion` usually is the same value as the `compileSdkVersion` — we update both of them at the same time to the same value.

Note that in Arctic Fox, you may see these values defined without the `Version` suffix, as `compileSdk`, `minSdk`, and `targetSdk`.

Version Code and Version Name

The `defaultConfig` closure has `versionCode` and `versionName` properties. These two values represent the versions of your application.

The `versionName` value is what the user will see for a version indicator in places like the Settings app and the Play Store. This can be whatever string you want, using whatever naming or numbering system that you want. However, for customer support purposes, you should have *some* system that varies by release, rather than using the same string all of the time.

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 Stuff in the `android` Closure

The `android` closure has a `testInstrumentationRunner` statement — we will explore that more [in an upcoming chapter, to see how testing works](#).

The `android` closure also has a `buildTypes` closure. This provides specific configuration for different “build types”, such as `debug` for development builds and `release` for production builds. The defaults provided in the starter project are fine for many basic apps.

The `compileOptions` and `kotlinOptions` closures indicate that we want the Java and Kotlin compilers to generate JVM 1.8 bytecode. While JVM bytecode has advanced a lot since then, 1.8 is the newest that Android supports at this time.

Libraries and Dependencies

Roughly speaking, the code and assets that make up an app come from three sources:

1. The source that you and people that you know are writing for this app.
2. The source that comes from your `compileSdkVersion`, representing the Android SDK that you are linking to.
3. Everything else, generally referred to as dependencies. These are libraries, written by Google, independent Android developers, major firms, and so on. Every modern Android app uses libraries, and bigger apps use more libraries.

From a pure technical standpoint, most dependencies are listed in `build.gradle` files in dependencies closures. We have seen two of these in this chapter.

One dependencies closure appears in the project-level `build.gradle` file, inside of a `buildscript` closure:

```
dependencies {  
    classpath "com.android.tools.build:gradle:7.0.2"  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.30"  
  
    // NOTE: Do not place your application dependencies here; they belong  
    // in the individual module build.gradle files  
}
```

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. Kotlin users will also use a Kotlin plugin.

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 one from the starter project:

```
dependencies {  
  
    implementation 'androidx.core:core-ktx:1.6.0'  
    implementation 'androidx.appcompat:appcompat:1.3.1'
```


REVIEWING YOUR GRADLE SCRIPTS

```
implementation 'com.google.android.material:material:1.4.0'  
implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
testImplementation 'junit:junit:4.+'  
androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
}
```

Here, there are three types of statements:

- `implementation` says “here is a dependency that I want to use for my actual app”
- `androidTestImplementation` says “here is a dependency that I want to use for testing”
- `testImplementation` says “here is a dependency that I want to use for... a slightly different type of testing”

(we will explore the differences between those in [an upcoming chapter](#))

Our starter project has eight statements in `app/build.gradle` attempting to pull in dependencies:

- Three are tied to testing. We will be discussing those [soon](#).
- Two are tied to Kotlin (`org.jetbrains.kotlin:kotlin-stdlib-jre7` and `androidx.core:core-ktx`) and will not be seen in pure-Java projects.
- The `constraintlayout` and `appcompat` dependencies are specifically part of Jetpack and are foundations for modern Android UI development. We will be spending quite a bit of the book going over what these offer and how you use them.
- The `com.google.android.material:material` contains the Material Components for Android, which we saw briefly back in [the chapter on resources](#)

Most likely, you will be adding other similar statements to this set, plus perhaps deleting ones that you are not actually using (e.g., the `fileTree()` one). We will see how to add other libraries as the book progresses, as many of the things that are common in Android app development require additional libraries.

Inspecting the Compiled App

Our starter project has a bit of code, a bunch of resources, a manifest, and Gradle build instructions. And, we can see the app running on a device or emulator.

However, Android is not running directly that bit of code and other stuff. Android Studio compiles the things in our project into something that Android then runs.

In this chapter, we will take a brief look at what Android Studio is building for us and how we can examine that output, beyond running it on Android.

What We Build

From the user's standpoint, we are building an app. Usually, that is true. The story can get complicated with specialized scenarios like Android Auto and Android Wear, but we can ignore those for the time being.

From a programming standpoint, what you are creating is an APK, perhaps by way of an app bundle.

APKs

An APK is the Android executable format. It is what gets installed on an Android device, and it is what the device runs.

An APK is a ZIP archive. You can unpack one using your favorite unZIP tool, though we will see a better option for many cases [later in this chapter](#).

The APK will contain:

- Your compiled code
- Your resources, usually in their own “compiled” format
- Your manifest, also in a “compiled” format
- Other miscellaneous files

This includes not only the things that you create yourself but also the stuff that you wind up using from libraries that are part of your app module.

App Bundles

The APK not only is what an Android device runs, but usually it is what Android app developers publish. You use Android Studio to create the APK from your project, and you upload that APK to the Play Store or other app distribution channels. The Play Store or other site then distributes the APK to users.

You will hear Google talk about “app bundles” as an alternative. App bundles are designed for large, complex apps, where only part of what goes in an APK will be relevant for any given device. For example, a major brand APK may have strings translated to many languages, but the user typically uses only one or two languages. Having translations in other languages just takes up disk space and adds no value to that individual user. App bundles allow Google to craft tailored APKs on your behalf that match the needs of individual devices and users.

However, now you are no longer in control over what is in your app. Google is. While Google likes to comment on what they can remove from your app (e.g., unnecessary languages), [they do not discuss whether they might *add* anything to your app or otherwise modify its behavior](#). That could be for their benefit or for the benefit of some third party.

You will need to decide for yourself whether the benefits of app bundles outweigh the risks. Note, though, that Google has stated that app bundles will be required for new apps being distributed through the Play Store starting in August 2021.

Where They Go

When you run your app in Android Studio, it creates an APK to install on the device. That APK will wind up inside of the module's `build/` directory. `build/` contains all sorts of outputs of the build process, the APK being chief among them:

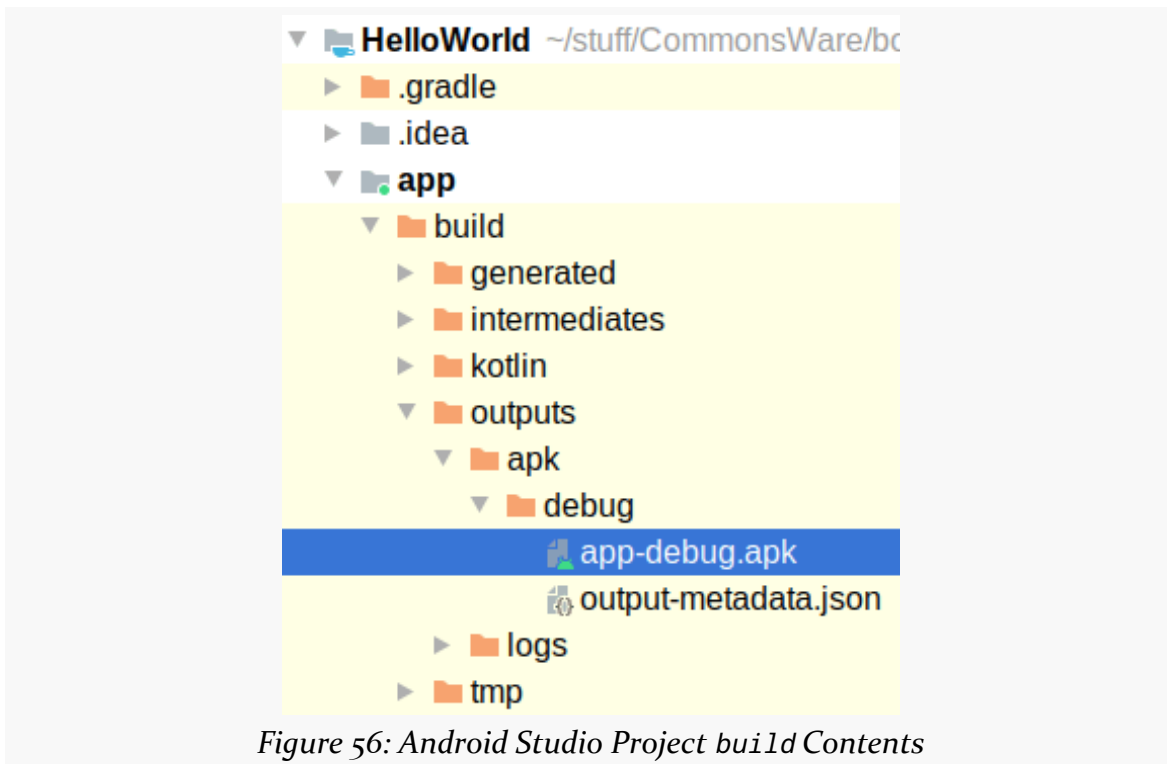


Figure 56: Android Studio Project build Contents

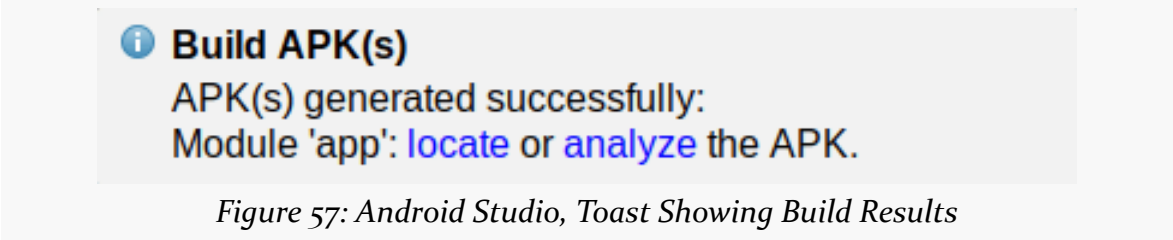
Specifically, when you start off, you will find the APK in `build/outputs/debug/app-debug.apk`. By default, the filename will be based on your Gradle module name, so an `app/` module winds up with an APK with `app` in the name. And, by default, you are creating a debug build — the sort of build that you as a developer will use — and so the APK winds up in a `debug/` directory and has `debug` in the name.

Building the APK

You can manually build the APK at any time through the Build menu in Android Studio. For example, you might want to do that before trying to [analyze the APK](#).

For creating an ordinary debug APK, choose `Build > Build Bundles(s) / APK(s) > Build APK(s)` from the Android Studio main menu. After a few moments, a “toast”-

style popup window should appear announcing the results:

A screenshot of an Android Studio toast notification. The toast has a light gray background and a blue information icon on the left. The text inside the toast reads: "Build APK(s)" in bold, followed by "APK(s) generated successfully:" and "Module 'app': locate or analyze the APK." where "locate" and "analyze" are blue and underlined, indicating they are clickable links.

i Build APK(s)
APK(s) generated successfully:
Module 'app': [locate](#) or [analyze](#) the APK.

Figure 57: Android Studio, Toast Showing Build Results

Analyzing the APK

One of the links in that toast is to “analyze” the APK.

Android Studio offers an APK Analyzer tool. Mostly, it exists to help developers see exactly what is in the APK, blended from all of the possible sources (your code, library code, etc.). For developers looking to reduce the size of the APK, the APK Analyzer is great for seeing where the space goes.

INSPECTING THE COMPILED APP

To bring up the APK Analyzer, either click that “analyze” link from a fresh build, or choose Build > Analyze APK at any time. The latter will bring up a typical file chooser dialog for you to navigate through the build/ directory to select the APK that you want to analyze:

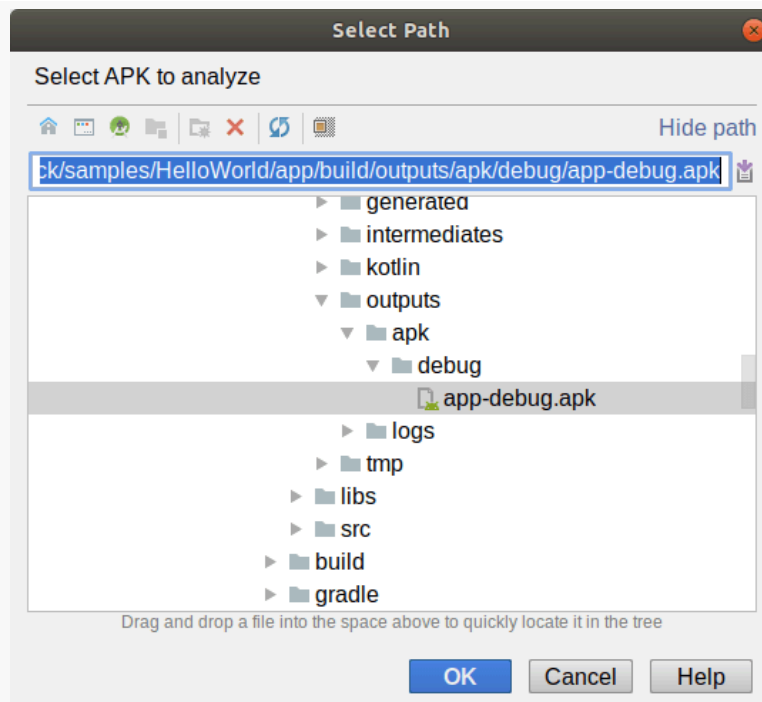
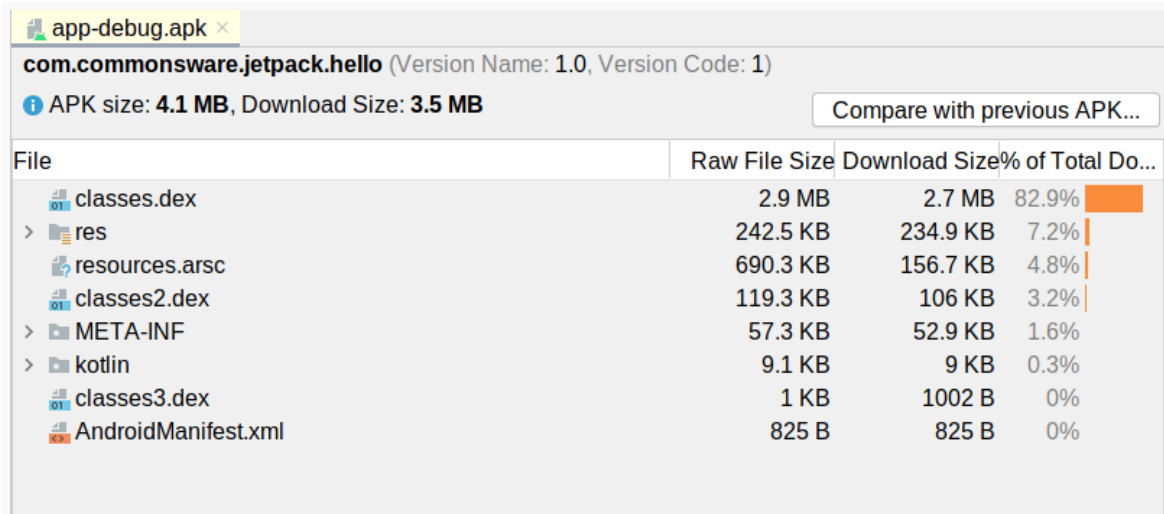


Figure 58: Android Studio, APK Selection Dialog

INSPECTING THE COMPILED APP

The APK Analyzer then opens in a fresh tab:



File	Raw File Size	Download Size	% of Total Do...
classes.dex	2.9 MB	2.7 MB	82.9%
> res	242.5 KB	234.9 KB	7.2%
resources.arsc	690.3 KB	156.7 KB	4.8%
classes2.dex	119.3 KB	106 KB	3.2%
> META-INF	57.3 KB	52.9 KB	1.6%
> kotlin	9.1 KB	9 KB	0.3%
classes3.dex	1 KB	1002 B	0%
AndroidManifest.xml	825 B	825 B	0%

Figure 59: Android Studio, APK Analyzer

The starter project's APK is a 3.5MB file. This may seem rather large for an app that does not really do much. That is because:

- Google's starter projects include a lot of [libraries](#), and
- A normal debug build does not eliminate unused stuff from those libraries

We will look into how to address that latter problem [much later in the book](#).

The table shows the breakdown of where the space goes. Of note:

- The .dex files — and, where relevant, the kotlin/ directory — represent your compiled code, both that you wrote, what was code-generated for you by Android Studio, and what you get from libraries
- The res/ directory and the resources.arsc file represent your resources
- The AndroidManifest.xml file is your manifest

We will explore these things much more [later in the book](#) and help you see how you might make your app smaller.

Touring the Tests

When we write Android apps, a chunk of our time is spent testing those apps. Some of that testing is manual: poking at the UI and seeing if everything works as expected. But some of that testing is automated, with test classes that test our “real” classes and confirm that everything is OK.

With that in mind, let’s take a look at the types of tests that we have in the starter project and how to run them.

Instrumented Tests

There are two major types of test in an Android app:

- Instrumented tests, which run in Android on a device or emulator
- Unit tests, which run on your development machine or similar places

Unit tests run much faster, but they cannot test as much, because they do not have access to everything inside of Android. For example, while we could test our ability to talk to a Web service from unit tests, we cannot test our ability to get GPS locations using Android APIs from unit tests. For those, we need instrumented tests. Similarly, most automated UI testing needs instrumented tests, as the Android UI system is only really available in Android.

Since they are more flexible, and since test speed only becomes a major issue with larger projects, let’s focus first on instrumented tests.

Where They Run

As noted above, instrumented tests will run on an Android device or emulator. For

your own personal test runs, you can use the same devices or emulators that you use for manually running the app.

Projects that employ continuous integration (CI) servers will need to configure them to support running tests on server-hosted emulators. Some hosted CI services — such as [CircleCI](#) — have that capability readily available to you. For self-hosted CI servers, there should be recipes available to teach you how to configure them for Android app testing.

What You Can Test

Because you are running the tests in an actual Android environment, you can test anything that you want. You have the full Android SDK at your disposal.

However, from a practical standpoint, there will be limits as to what you can test:

- Emulators do not emulate everything about hardware. For example, you will not be able to test readings that you get from nearby cell towers, as an emulator is not in communication with any actual cell towers.
- You want your tests to be repeatable. Hence, even on hardware, you may need to limit testing what you really get from the hardware, as you do not control that hardware and what it might return. For example, while in theory you could test getting actual location data via GPS from a device, you cannot guarantee the precise values that will get returned, as GPS is inexact by its very nature.
- Any given device has one set of hardware characteristics. Any given emulator will mimic one set of hardware characteristics. Testing things that vary based on hardware characteristics will require multiple test runs across a fleet of devices or emulators that will reflect the varying characteristics.

What the Starter Project Has

The starter project not only has a “hello, world” sort of UI for you, but it has a similar instrumented test set up, ready for you to run.

The `androidTest` Source Set

As we saw earlier in the book, instrumented test code resides in an `androidTest/` directory. This is a peer to the `main/` directory that contains your “real” application code. `androidTest/` is a “source set” that will be used only when running

instrumented tests. The stuff in the `androidTest/` source set will *not* be included in your app when you ship it.

The Test Class

Inside of there you will find a `java/` directory, with a Java package matching the application ID of your app, and an `ExampleInstrumentedTest` Java or Kotlin file.

When you create a new project, and you choose whether or not to have Kotlin support, that choice will determine not only whether your `MainActivity` is in Java or Kotlin, but also whether your test code is in Java or Kotlin.

The Kotlin class is fairly short:

```
package com.commonware.jetpack.hello

import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val appContext = InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals("com.commonware.jetpack.hello", appContext.packageName)
    }
}
```

The Java equivalent is not much longer:

```
package com.commonware.jetpack.hello;

import android.content.Context;
import androidx.test.platform.app.InstrumentationRegistry;
```

```
import androidx.test.ext.junit.runners.AndroidJUnit4;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

/**
 * Instrumented test, which will execute on an Android device.
 *
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
 */
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Test
    public void useAppContext() {
        // Context of the app under test.
        Context appContext =
            InstrumentationRegistry.getInstrumentation().getTargetContext();
        assertEquals("com.commonware.jetpack.hello", appContext.getPackageName());
    }
}
```

The Annotations

The Java and Kotlin editions of the test class are equivalent, other than language syntax.

Both have a single class, named `ExampleInstrumentedTest`, annotated with a `@RunWith(AndroidJUnit4::class)` annotation. Android presently uses [JUnit 4](#) for instrumented tests. This annotation tells JUnit — and, more importantly, some Android Studio stuff for running tests — that this class contains test code that should be run as part of an instrumented test.

Both editions of `ExampleInstrumentedTest` have one method (or function, in Kotlin). It is called `useAppContext()`, and it is marked with the `@Test` annotation. A test class can contain one or more of these `@Test` methods/functions. When it comes time to run the tests, Android Studio will:

- Create an instance of your test class
- Call one of the `@Test` methods/functions on that instance
- Create another instance of your test class
- Call another of the `@Test` methods/functions on that new instance
- And so on, until all of the `@Test` methods/functions have been executed

The Test Code

So... what is `useAppContext()` testing?

In truth, it is not testing very much.

We will explore what a Context is [a bit later in the book](#). For the moment, take it on faith that this code is:

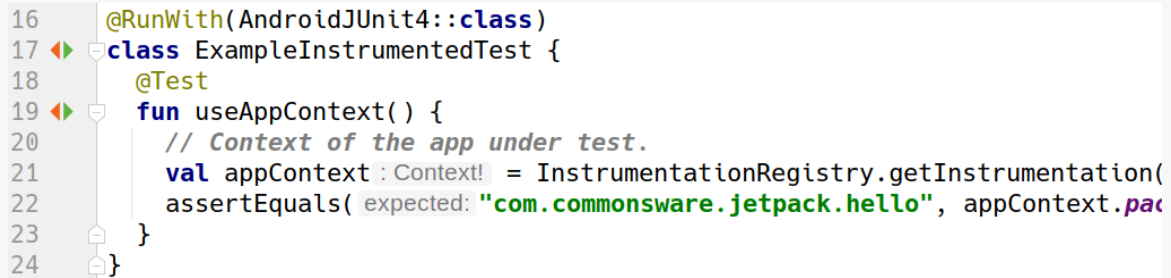
- Finding out what our application ID is, by calling `getPackageName()` on a Context
- Confirming whether it matches the expected value of `com.commonware.jetpack.hello`

`assertEquals()` is supplied by JUnit 4 and will fail the test if the two values are not equal.

We will explore much more about JUnit 4 and how to write more elaborate tests [much later in the book](#).

How You Run Them

For a single test method or function, you will notice a triangle-pair “run” icon in the “gutter” area of the code editor:



```
16 @RunWith(AndroidJUnit4::class)
17 class ExampleInstrumentedTest {
18     @Test
19     fun useAppContext() {
20         // Context of the app under test.
21         val appContext : Context! = InstrumentationRegistry.getInstrumentation().context
22         assertEquals( expected: "com.commonware.jetpack.hello", appContext.packageName )
23     }
24 }
```

Figure 6o: Android Studio, Showing Instrumented Test in Kotlin

Clicking that will allow you to run that individual test method/function. Or, optionally, you will be able to [debug that test method](#).



Figure 61: Android Studio, Showing Pop-Up Menu for Running a Test Function

Similarly, there is a “run” icon in the gutter next to the class name, to run all of the test functions in the Kotlin test class. For a Java class, the class-level test icon is a “double-run” pair of overlapping green triangles:

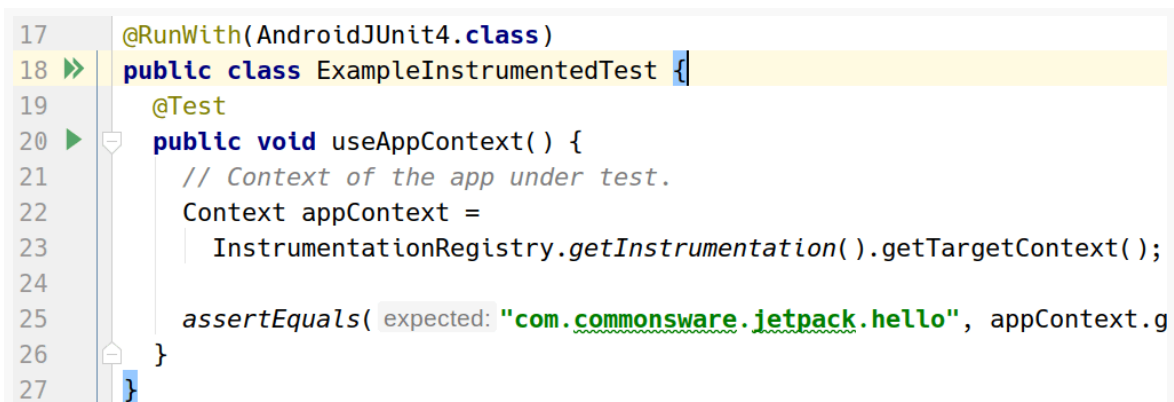


Figure 62: Android Studio, Showing Instrumented Test in Java

(Why is there a difference? [Ask Google](#).)

TOURING THE TESTS

For smaller projects, those may suffice. For larger projects, you can set up a custom “run configuration” that can run all of the instrumented tests in your project, for example. We will see this [much later in the book](#).

What the Test Results Look Like

When you run a test method or a test class, rather than focusing on the output in the emulator or device, you will focus instead on the “Run” view in Android Studio. This will show you which tests succeeded and which tests failed.

The nice people who created Android Studio elected to write tests that succeed, and Android Studio’s output will reflect that:

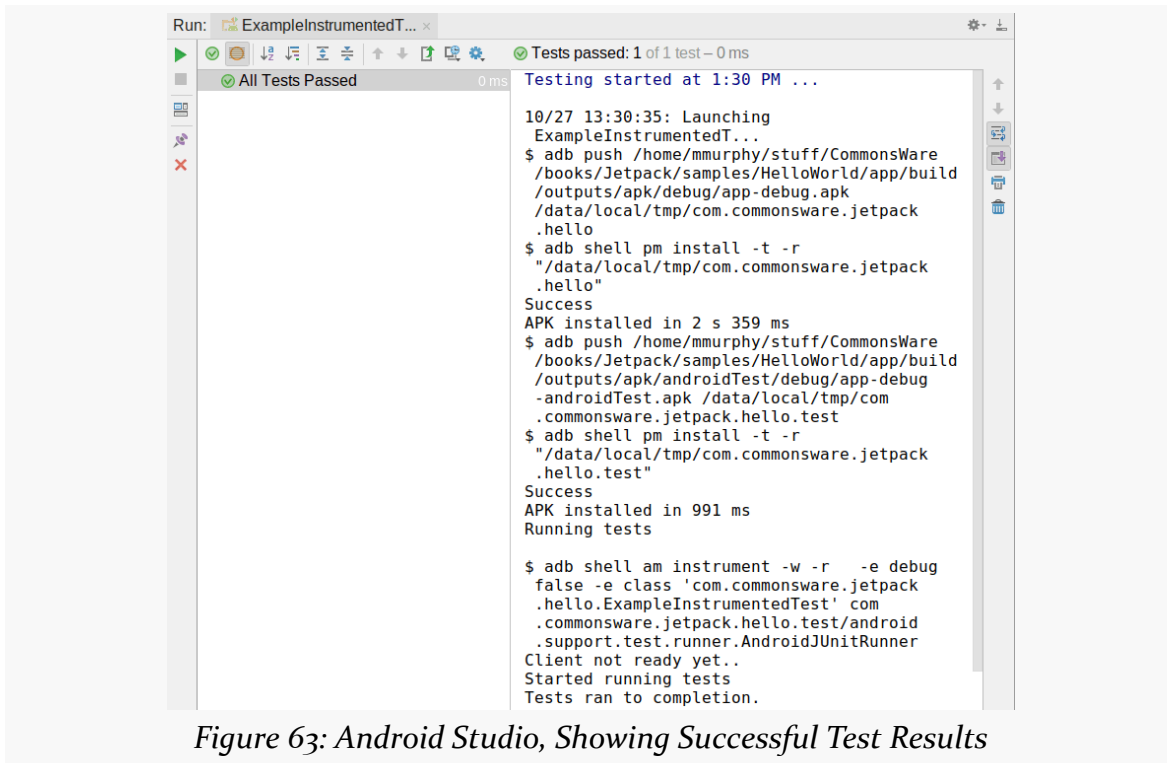


Figure 63: Android Studio, Showing Successful Test Results

Note the “All Tests Passed” message with the green checkmark-in-circle icon.

TOURING THE TESTS

If a test fails — such as a modified version of the sample’s test class that compares the application ID to this.is.wrong — you will see the failed test in a tree on the left and details of what went wrong on the right:

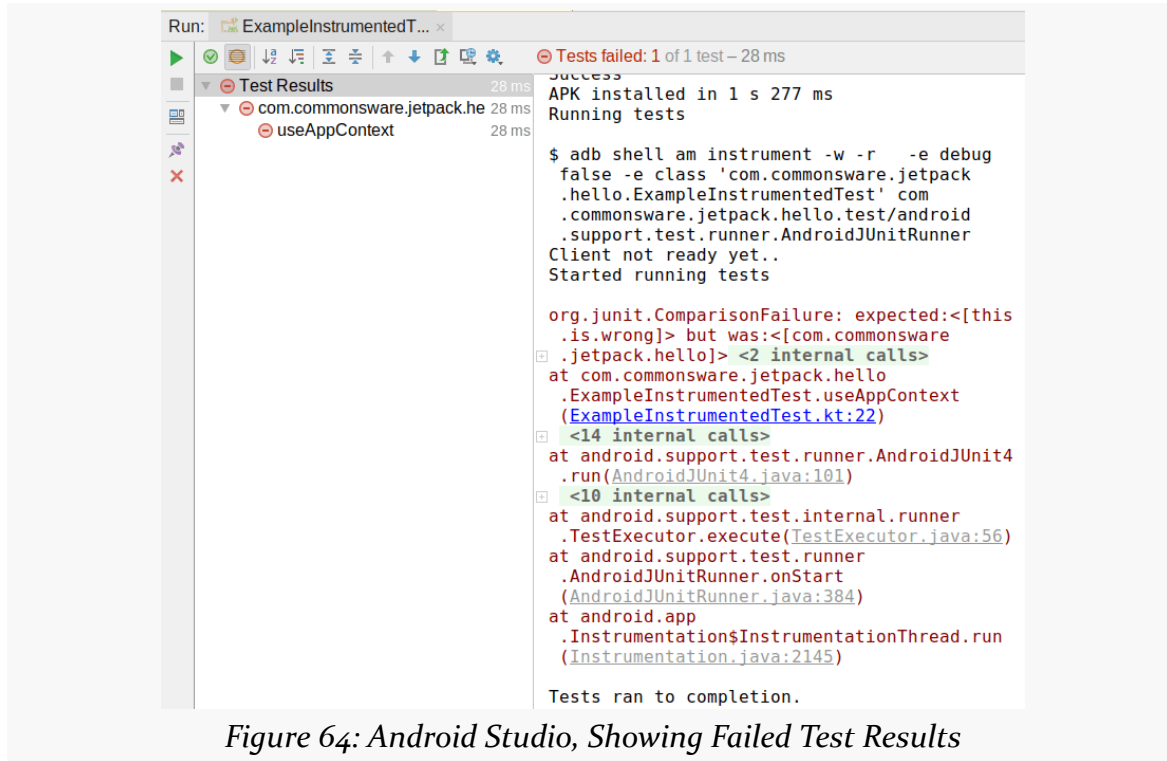


Figure 64: Android Studio, Showing Failed Test Results

JUnit assertions, such as `assertEquals()`, will provide some details as to what went wrong, shown in a stack trace in the test output:

```
org.junit.ComparisonFailure: expected:<[this.is.wrong]> but
was:<[com.commonware.jetpack.hello]>
```

Here, we see that we expected `this.is.wrong`, but we instead got `com.commonware.jetpack.hello`, and so the test failed.

About That `testInstrumentationRunner`

Back in [the chapter on Gradle](#), we saw this line in the `defaultConfig` closure:

```
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
```

The `testInstrumentationRunner` indicates what code should be used to execute the

JUnit tests themselves. The runner shown here is the standard runner for instrumented tests. Various third-party testing tools might have you replace this value with some class from their library, so this is not *always* the test runner that we use.

The androidTestImplementation Dependencies

Such a library would also show up as an `androidTestImplementation` dependency in the list of dependencies for the module. We have a couple of those already:

```
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
```

The one that is really required is `androidx.test:runner`. This is the library that supplies the `AndroidJUnitRunner` class and other core classes for writing and running our instrumented tests.

The other one is `androidx.test.espresso:espresso-core`. This is the core of Espresso, a powerful library for writing GUI tests. However, our existing instrumented test does not actually *use* Espresso, so this particular dependency is unnecessary at the moment. We will cover Espresso [much later in the book](#).

Unit Tests

The other type of tests are unit tests. They too use JUnit4, which causes some confusion when developers try to determine the difference between the instrumented tests and unit tests.

Where They Run

Unit tests run in Windows, macOS, or Linux, not Android. Unit tests are the same sorts of tests that you would run in an ordinary Java or Kotlin project that had no ties at all to building Android apps.

You can run unit tests on your own development machine. You can also arrange to run unit tests on a CI server or some similar environment.

Because unit tests avoid a lot of the work in getting code over into an Android environment, unit tests can run more quickly. Plus, a development machine or CI server is likely to be faster than Android hardware, and even the Android emulator

adds some amount of runtime overhead.

What You Can Test

You can test your non-Android business logic fairly easily, just using standard JUnit tests. However, any code that uses Android-related classes is going to have a problem, though.

You can get past this somewhat by the use of mocking engines, such as Mockito. They allow you to create “fake” Android objects based on real Android classes, where you teach the mocks how to respond to particular method calls.

What the Starter Project Has

Our starter project has a similar set of stuff for unit tests as it did for instrumented tests, with some modifications.

- Whereas instrumented tests go in an `androidTest/` source set, unit tests go in a `test/` source set.
- In there, as with the `androidTest/` source set, there is a `java/` directory and a `com.commonware.jetpack.hello` package. And, once again, there is a single class, though this time it is named `ExampleUnitTest`.

The Annotations and the Test Code

Once again, the class will either be in Java or Kotlin:

```
package com.commonware.jetpack.hello

import org.junit.Test
import org.junit.Assert.*

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

TOURING THE TESTS

```
package com.commonware.jetpack.hello;

import org.junit.Test;

import static org.junit.Assert.*;

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
 */
public class ExampleUnitTest {
    @Test
    public void addition_isCorrect() {
        assertEquals(4, 2 + 2);
    }
}
```

Unit test classes do not normally get a `@RunWith` annotation. However, we still put `@Test` annotations on the methods or functions that contain the tests to be run.

In both Java and Kotlin, the tests use JUnit's `assertEquals()` to compare two values for equality. This time, the test is to confirm that $2 + 2$ equals 4. This test usually succeeds.

How You Run Them

You can run unit tests in much the same way as you run instrumented tests:

- Click the “run” icon next to a test method/function to run it alone
- Click the “run” icon next to a test class to run all of its test code
- Set up a “run configuration” to run all of your unit tests, which we will see [much later in the book](#)

What the Test Results Look Like

When you run unit tests, you will not be prompted for a device or emulator to run them on. Unit tests run directly on your development machine, not on an Android device or emulator.

TOURING THE TESTS

They will run much faster, and they will provide similar output in the “Run” view of Android Studio:

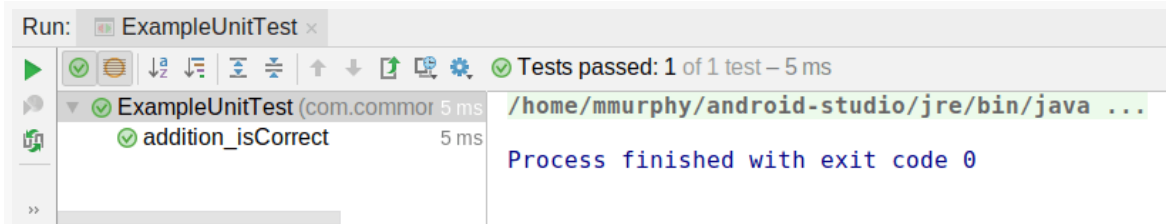


Figure 65: Android Studio, Showing Results of Successful Unit Tests

The testInstrumentation Dependencies

Any testImplementation statement in your dependencies in your module's build.gradle file will be compiled into your tests and available for use. The one default testImplementation dependency is junit:junit, which pulls in the JUnit classes.

Introducing Jetpack

This book is titled “Elements of Android Jetpack”. Yet, we have not yet discussed Jetpack at all. So, it’s about time that we saw what Jetpack is and what it has to do with Android app development, your future projects, and the rest of this book.

What, Exactly, is Jetpack?

Jetpack... is a brand name.

In 2018, Google elected to use the term “Jetpack” to refer to a seemingly-random collection of Android technologies and tools. In many respects, the only things that those technologies and tools have in common is:

- They are involved in Android app development
- Google thinks that they are part of Jetpack

There is no singular “Jetpack” tool or library. In fact, outside of the cover and the contents of this chapter, “Jetpack” will not be used much as a term in this book, as it focuses on what is *in* Jetpack more so than Jetpack itself.

Um, OK, So, What’s the Point?

The first Android devices shipped to the public in October 2008. Public Android app development had been going on for about a year prior to this, with the first preview releases and Android SDK bits available in 2007. Google had been working on Android apps for a while prior even to that preview release.

In short, Android app development has been around for a while.

Along the way, many techniques and approaches were tried and discarded. Many programming patterns were applied, rejected, and replaced. Many UI designs were rolled out to great fanfare, only to be shunted aside in favor of yet new UI designs. And so on. Android app development has a lot of history, and while some of that history remains relevant today, a lot of it is just baggage.

Jetpack is a way of thinking about Android app development that dumps some of that baggage.

The focus of Jetpack — and this book by extension — is on the current recommended practices in Android. We try to ignore most of the history, or at least put it in its appropriate place. Instead, we try to help you build using up-to-date approaches, bypassing the approaches of yesteryear.

Key Elements of Jetpack

Not everything in the Jetpack is of equal importance. This book focuses on those elements that are commonplace for most ordinary Android apps or are tied closely to other commonplace Android app development steps.

AppCompat

As was mentioned earlier in this chapter, Android is over 10 years old. Android changes with every release.

However, Android devices do not get very many OS updates from their manufacturer. Even the best might only get updates for 2-3 years, and there are plenty that *never* get an update. As a result, Android users use a wide range of Android OS versions. While the changes from release to release may be small or may be large, the combined changes from older versions of Android to newer ones can be vast.

AppCompat tries to help. It gives us an API for our activities and fragments that resembles the latest-and-greatest version of Android. When your app runs on older devices, AppCompat tries to fill in the gaps of UI functionality where it can. While using AppCompat makes your APK a lot larger and makes app development more confusing, many developers are grateful for the backwards-compatibility that it offers.

We will dive more deeply into what AppCompat is, and its relationship to your app,

[in an upcoming chapter](#).

Fragment

A layout does not exist on its own. Something has to arrange to use a layout resource and show it on the screen. We saw in the previous chapters that an Activity can be used for that, and we will start there for seeing how to build Android user interfaces.

However, in many cases, you will wind up using fragments for loading and displaying your layout resources. Fragments are wrappers around layout resources and their widgets. Many apps might consist of a single activity, with each different “screen” being represented by a fragment that uses associated layout resources and code. The user, as part of using the app, will switch from fragment to fragment as needed — for example, clicking a button might cause another fragment to be shown on the screen.

We will look at fragments [a bit later in the book](#).

Navigation

While some apps might have just one screen of information, most have more than one. Users click on widgets and are taken to other screens. Users then press the BACK button and return to the screen they had been on originally. And so forth.

We have had ways of accomplishing this sort of navigation since Android’s introduction. However, it involves Java/Kotlin code, and sometimes quite a bit of it, for complex interactions. Google keeps trying to move some of that sort of work into resources, where it can.

The Navigation library provides a way to declare how to move from screen to screen via resources, with a new editor in Android Studio to assist in defining those resources and depicting your navigation flow.

We will explore this Navigation library as part of our discussion of [fragments](#).

Lifecycles, ViewModel, and LiveData

Most Android devices are phones or tablets. These can be held in portrait or landscape. Well-written apps appear to seamlessly transition between the two as the user rotates the screen.

In reality, screen rotations and other types of “configuration changes” are among the most annoying aspects of Android app development. It turns out that our activities and fragments have “lifecycles”, and they come and go not only based on user navigation but also these configuration changes. Keeping our state as the user rotates the screen is important, as users get irritated if we lose stuff.

To that end, Google nowadays offers two Jetpack pieces to help with this:

- Special code for dealing with lifecycles
- A `ViewModel` class that helps maintain our state across configuration changes

For some developers, configuration changes are the most annoying thing in Android app development. For others, it is threads. We need to do work on background threads to keep our UI “responsive”, accepting and processing user input while that background work is ongoing. This is a pain to manage in its own right, and configuration changes make it that much more difficult. `LiveData` objects — held by our `ViewModel` — make it somewhat easier to do work in background threads while not losing track of that work if the user rotates the screen, for example.

In this book, we will explore [ViewModel and lifecycle handlers](#) as well as [LiveData and threads](#).

Room

Many apps need a database on the device. Sometimes, that is the “system of record”, as the data is only on the device. Sometimes, the database serves as more of a local cache, for data that we obtained from a server or for data that we need to get to a server.

Android has had a relational database, called `SQLite`, in its SDK since the beginning. However, the standard Android APIs for working with this database are a bit crude by modern standards. To that end, Google has created Room, a lightweight object wrapper around `SQLite` databases, to simplify your database I/O. Room also works nicely with `LiveData`, to help you do your database I/O on background threads.

We will explore the basics of Room [much later in the book](#).

WorkManager

Many times, the work that we need to do on background threads has to be performed in near-real time. But sometimes the work that we need to do can

happen totally independently from what the user may (or may not) be doing in our UI. So while a “pull to refresh” UI operation requires us to do a refresh right now, a periodic refresh from the server might happen every few hours, even if the user is not in the app right now.

While there have been many solutions over the years for this problem, Google is steering us towards `WorkManager` right now. We can teach `WorkManager` about background work to be performed, then schedule that work to occur. That schedule might be based in part on time (“do it soon”, “do it every few hours”, etc.). That schedule might be based in part on the state of the device (“do it when the device is on a charger”, “do it when the device is on WiFi”, etc.). Then, `WorkManager` will arrange to do the work, even if the user leaves our UI.

We will examine `WorkManager` [towards the end of the book](#).

Android KTX

The Android SDK is based on Java, and it has a fairly typical Java API, albeit one with a variety of patterns, based in part on the age of the code.

Kotlin is Google’s recommended language for Android developers going forward, with Java being relegated to the SDK itself. This is similar to how Apple steers developers towards Swift, with Objective-C being used only sporadically.

One of Kotlin’s features is the “extension function”. This is a way for a library to add functions (Java methods) to somebody else’s Java/Kotlin class. For developers using the class, the extension functions are seamlessly integrated with the functions that were part of the class from the outset.

Android KTX is a collection of such extension functions, designed to make common aspects of the Android SDK a bit easier to use.

We will see Android KTX code sprinkled throughout this book.



You can learn more about extension functions in the “Extension Functions” chapter of [Elements of Kotlin](#)!

Testing

No software project is complete without testing. We have already seen how Android projects can have instrumented tests and unit tests. We are given stub tests when we create our project, just to get us going, but we will need to write a lot more tests before we are done.

We will be exploring testing more [later in the book](#).

Compose

The most recent major addition to the Jetpack is [Jetpack Compose](#).

Jetpack Compose is a new way of implementing user interfaces in Android. In effect, it is a Kotlin domain-specific language (DSL) for implementing user interfaces. It replaces things like layout resources with pure Kotlin code. In the coming years, it is likely that Compose will become a very important part of the Android development ecosystem.

However, Compose only received a stable 1.0.0 release in July 2021. While it can be used for shipping apps, it does have a fair number of bugs which may impede your progress. In addition, only a tiny fraction of the world's Android UIs are created using Compose today, and only a tiny fraction of the world's knowledge of Android app development centers around Compose.

This book does not cover Compose. You will want to keep an eye on how rapidly Compose gets adopted by developers and make plans to learn Compose at the appropriate time.

What Came Before: the Android Support Library

Many of the capabilities described here as being part of Jetpack have existed for a while, in some cases for many years.

The biggest change with Jetpack — besides the branding — is the set of libraries that we use. The Jetpack libraries are all “AndroidX” libraries, where we will be referring to libraries and Java packages that have `androidx` in their names.

Prior to Jetpack, we had a set of libraries that evolved over the years, with a variety of naming schemes for libraries and Java packages. Much of what is in the Jetpack

originated in what we called the Android Support Library. Some came from the Architecture Components, which was another family of libraries.

Since Google debuted Jetpack in May 2018, anything written prior to that would be referring to these older library collections. Most of what was in those libraries has been replaced by AndroidX equivalents, though not everything. This will cause some amount of confusion, particularly since a single Android project cannot easily combine the older libraries (Support Library and Architecture Components) and the AndroidX libraries.

As a result, particularly as you learn Android app development, it will help if you know the dates for your educational materials and online resources. Whenever you see a date prior to May 2018 — and even for some stuff after that date — just bear in mind that you may need to convert some older library and Java package names into their AndroidX equivalents.

Introducing the Sampler Projects

It is time to move past what Android Studio gives you by default for a new project and start seeing how we can add our own stuff to these projects.

To that end, this book has a pair of Android Studio projects that contain the code samples that will be shown in the remainder of the book. You are welcome to download one or both of these projects to be able to run the samples on your own devices or emulators.

The Projects

The two projects are very similar, in terms of the Android SDK features that they demonstrate. The primary difference: one is in Kotlin, and the other is in Java. The resources, manifests, and much of the Gradle build files will be the same, but the source code will be in the programming language for the project:

- `Sampler` is in Kotlin
- `SamplerJ` is in Java

This way, you can choose which project to work with, based on which programming language you want to use for your initial Android work. In general, the rest of the book will show you both the Java and the Kotlin editions of the code, so you can compare and contrast the two languages and Android's support for each. In particular, if you are new to Kotlin, these corresponding samples can help you see how Java approaches get translated into Kotlin. Note that there will be a few samples in the latter half of the book that are Kotlin-only.

These two projects are hosted on [GitLab](#):

- The Java project is at: <https://gitlab.com/commonsguy/cw-jetpack-java>
- The Kotlin project is at: <https://gitlab.com/commonsguy/cw-jetpack-kotlin>

Getting a Sampler Project

Relevant portions of the Sampler projects are shown directly in this book, including links to the GitLab Web site where you can view the full source files. So, if all you want to do is to read the code, you do not need anything more than this book and a Web browser.

If, on the other hand, you want to run the sample code, or perhaps modify it as part of your own experiments, you will need to get one or both of the Sampler projects onto your development machine and into Android Studio. You can do that completely from inside Android Studio if you wish, or you can get the source code separately and then import it into Android Studio.

Direct From Android Studio

Android Studio has built-in support for Git. You can download and import a Git project into Android Studio via File > New > Project from Version Control from an existing Android Studio window.

If you are at the Android Studio “welcome” dialog, click “Check out project from Version Control” and choose Git from the menu. If you already have a project open in Android Studio, choose File > New > Project from Version Control from the main menu.

This will bring up a “Get from Version Control” dialog box:

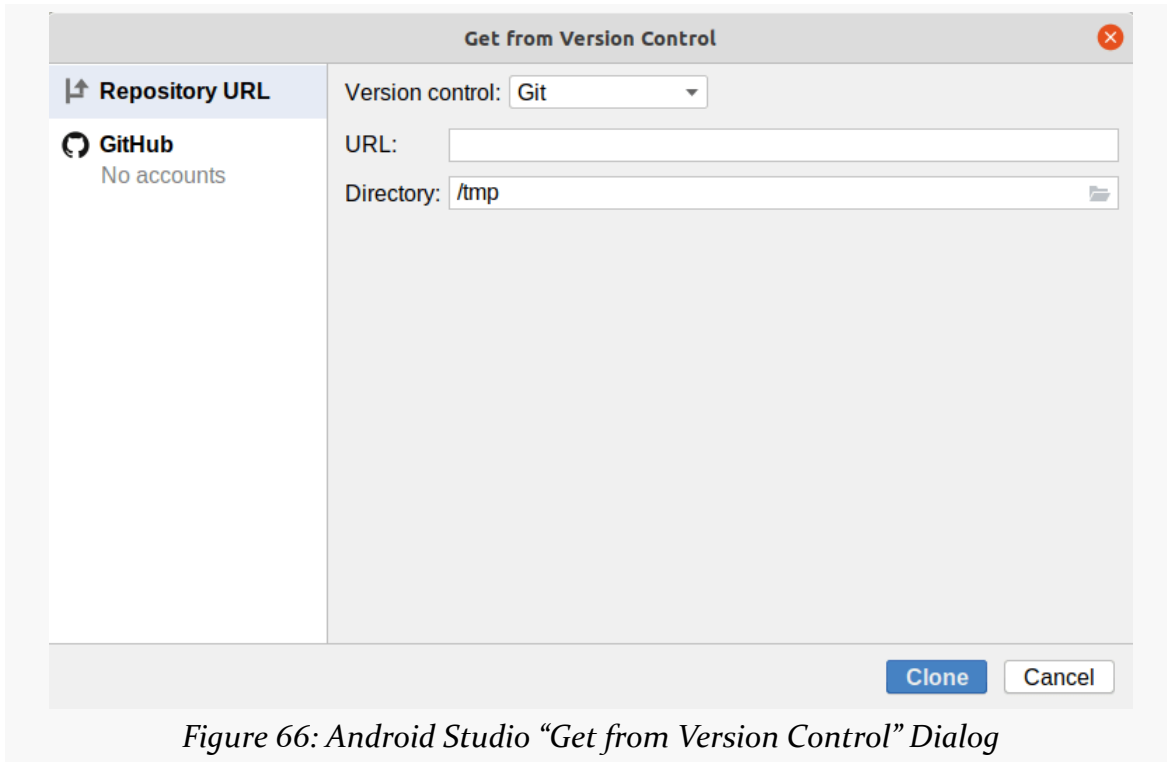


Figure 66: Android Studio “Get from Version Control” Dialog

Choose “Git” in the “Version control” drop-down. Then, in the “URL” field, fill in the Git URL for the project that you want to clone:

- Java: <https://gitlab.com/commonsguy/cw-jetpack-java.git>
- Kotlin: <https://gitlab.com/commonsguy/cw-jetpack-kotlin.git>

In the “Directory” field, you can provide the path to the directory on your development machine where you want to put the source code. This needs to be a new empty directory, but otherwise it can be wherever you want. The folder icon next to that field will bring up a directory chooser window to help you pick or create the directory that you want to use.

Then, click the “Clone” button to download the code and begin importing it into Android Studio. After a short while, your project will then be opened into an Android Studio window and be ready for use.

Manually

If you prefer to use other tools to work with Git repositories, you are welcome to do that too.

Clone or Download

Using your favorite Git client, clone the repository from its associated URL:

- Java: <https://gitlab.com/commonsguy/cw-jetpack-java.git>
- Kotlin: <https://gitlab.com/commonsguy/cw-jetpack-kotlin.git>

Import

Then, in an existing Android Studio window, you can choose File > New > Import Project to bring up a file/directory chooser. Alternatively, if you are at the welcome dialog, choose “Import an existing project” to bring up that same chooser.

Select the directory into which you cloned or unZIPped the Git repo, then click OK. It should open up right into an Android Studio window.

The Modules

Each project has the same set of modules (mostly):

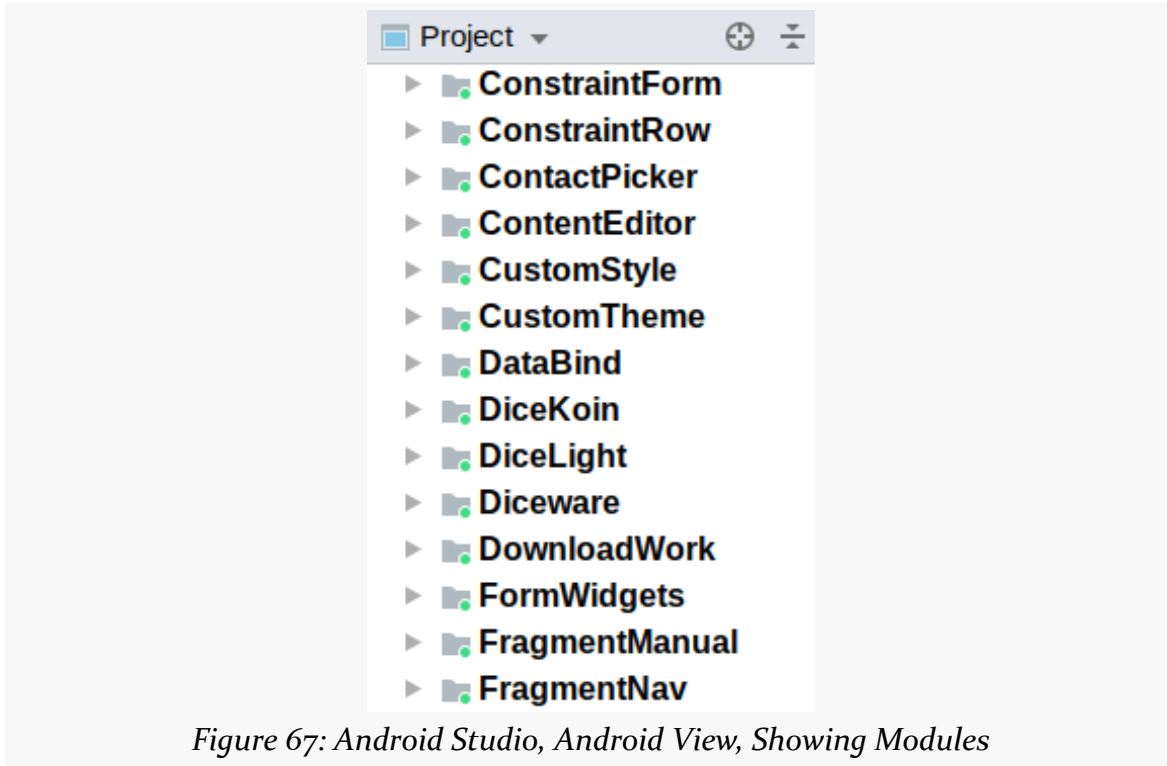


Figure 67: Android Studio, Android View, Showing Modules

Each module represents one sample app. Some chapters will use a few sample apps. Some sample apps will be used across multiple chapters. Each chapter will point out the modules that it is showing you, so you can synchronize your IDE with what the book is covering.

Running the Samples

Each module is ready to run!

In the toolbar, there is a drop-down list:



Figure 68: Android Studio, Run Configurations Drop-Down

Part Two: Creating an App's UI

Starting Simple: TextView and Button

Android offers a wide assortment of UI elements. Some are part of the Android SDK. Others come from third-party libraries. If the UI element is something commonplace, seen in multiple platforms or GUI environments, then it is likely that Android has an analogue of it somewhere.

But, as the saying goes, we must learn to walk before we can run.

So, in this chapter, we will explore the basics of Android's widget-based UI system by looking at two extremely common widgets: TextView and Button.

First, Some Terminology

Let's start off by defining some terms as they will be used in Android app development — and, by extension, in this book.

Widgets

Wikipedia has [a nice definition of a widget](#):

A graphical widget (also graphical control element or control) in a graphical user interface is an element of interaction, such as a button or a scroll bar. Controls are software components that a computer user interacts with through direct manipulation to read or edit information about an application.

(quote from the 14 May 2020 version of the page)

STARTING SIMPLE: TEXTVIEW AND BUTTON

Take, for example, this Android screen:

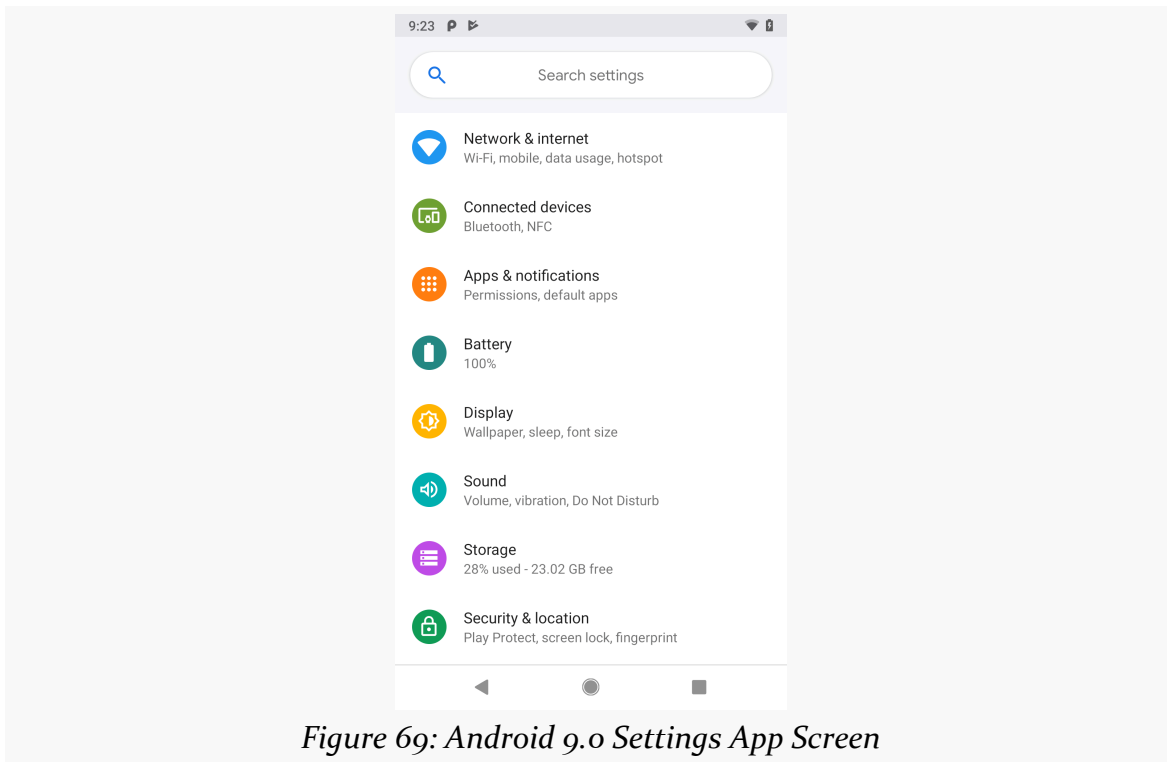


Figure 69: Android 9.0 Settings App Screen

Here, we see:

- a vertically-scrolling list, with rows containing two pieces of text and an icon
- a search field, with a magnifying glass icon and a space to type in a search term, contained in a rounded rectangle

Everything listed above is made of widgets. The user interface for most Android screens 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.

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

From a coding standpoint, widgets extend from a base class named `android.view.View`.

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 are 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.

Android supplies a handful of containers, designed to handle most common scenarios. However, we will use one most of the time: `ConstraintLayout`. This particular container gives us a flexible set of rules to decide where to place widgets within the screen. We will explore `ConstraintLayout` in [an upcoming chapter](#).

From a coding standpoint, containers extend from a base class called `android.view.ViewGroup`. `ViewGroup` itself inherits from `View`, and so (usually) things that we can do to a widget we can also do to a container.

Attributes

Widgets have attributes that describe how they should look and behave. In a layout resource, 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/Kotlin code.

For example, widgets and containers have a “visibility” attribute. This is set by:

- `android:visibility` attribute in a layout resource
- `setVisibility()` in Java
- Assigning a value to the `visibility` property in Kotlin

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. 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 a variety of popular widgets and the most commonly-used attributes on those widgets.

Widget IDs

Most widgets and containers in a layout resource will have an ID associated with them, by means of an `android:id` attribute. This serves two roles:

1. It allows you to reference that widget from within the layout file, for relative positioning rules (e.g., put this Button below that other Button)
2. It allows you to reference that widget from Java/Kotlin code

For example, here is a TextView widget with an ID of `@+id/hello`:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="@dimen/hello_padding"
    android:text="@string/hello"
    android:textSize="@dimen/hello_size"
    android:textStyle="bold" />
```

(from [SimpleText/src/main/res/layout/activity_main.xml](#))

Usually we use `@+id/...` as the `id` value, where the `...` represents your locally-unique name for the widget. For example, you might have a widget whose ID is `@+id/postalCode` or a container whose ID is `@+id/addressInfo`. You may find `android:id` values that lack the `+` sign (e.g., `@id/postalCode`) — usually this means that somewhere else in the same layout resource, there will be one for the same ID value, but with the `+` sign.

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. We will see attributes like `android:layout_height` and

`android:layout_width` that can be used by a widget to suggest sizing rules to its parent.

Let's look again at that `TextView`:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="@dimen/hello_padding"
    android:text="@string/hello"
    android:textSize="@dimen/hello_size"
    android:textStyle="bold" />
```

(from [SimpleText/src/main/res/layout/activity_main.xml](#))

Here, we have both `android:layout_height` and `android:layout_width` set to `wrap_content`. `wrap_content` tells Android “please size this based on whatever is inside of it”. For example, with a label (`TextView`), “whatever is inside of it” is the text that should be displayed. For a container (e.g., `ConstraintLayout`), “whatever is inside of it” is the set of children that it manages.

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 designed to help prevent widgets from running right up next to each other, so they are visually distinct. We will see attributes like `android:layout_margin` for specifying margins.

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. We will see attributes like `android:padding` for specifying padding.

Hey, What Are Those `@dimen` Things?

You will notice that the `android:padding` attribute has a value of `@dimen/hello_padding`. That is a reference to a dimension resource. Like string and color resources, dimension resources reside in files in `res/values/`, typically named `dimens.xml`. There, you give the dimension a name and a corresponding value:


```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="hello_size">24sp</dimen>
  <dimen name="hello_padding">8dp</dimen>
</resources>
```

(from [SimpleText/src/main/res/values/dimens.xml](#))

Here, we have two dimension resources, `hello_size` and `hello_padding`.

The value of a dimension resource is made up of two pieces:

- An integer or floating-point number, and
- A unit of measure

`hello_padding` is defined as `8dp`. `dp` is short for “density-independent pixels”. There are 160 `dp` in an inch. `dp` is the typical unit of measure for most things in Android.

However, `hello_size` is `24sp`. `sp` is short for “scaled pixels”. One `sp` is the same size as one `dp` for the default font scale. However, users can go into the Settings app and change their font scale, to make text bigger or smaller. `sp` takes that into account. So, if you use `sp` dimensions for your text size — say, in a `TextView` widget — your text will also adjust based on the user’s chosen font scale.

There are other units of measure that you could use but are almost never used:

- `in` for inches
- `mm` for millimeters
- `px` for hardware pixels

In particular, trying to use `px` for a dimension will result in compiler warnings. Using `px` does not take screen density differences into account, so a dimension measured in `px` will be different sizes on different devices. This is almost never what you want.

Introducing the Graphical Layout Editor

If you open a layout resource in Android Studio, you will see one of two perspectives: XML, or a drag-and-drop graphical editor.

STARTING SIMPLE: TEXTVIEW AND BUTTON

Towards the upper-right of the layout resource editing tab, you will see that it has a few toolbar buttons:

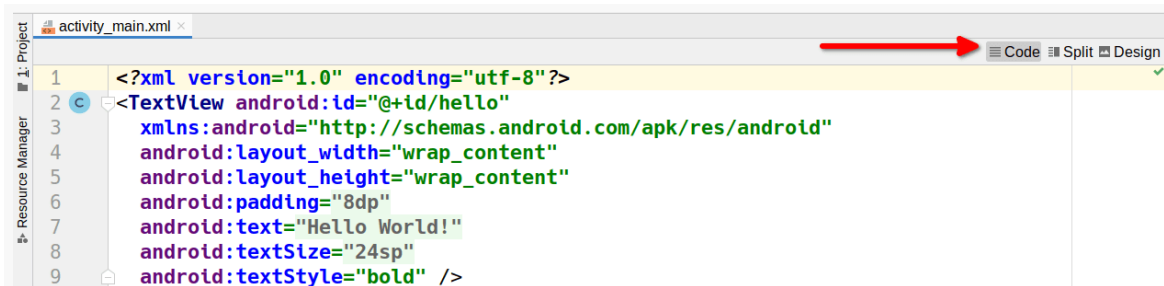


Figure 70: Android Studio Graphical Layout Editor, with Toolbar Buttons Highlighted

One will be selected at any point in time. The “Design” one represents the graphical layout editor:

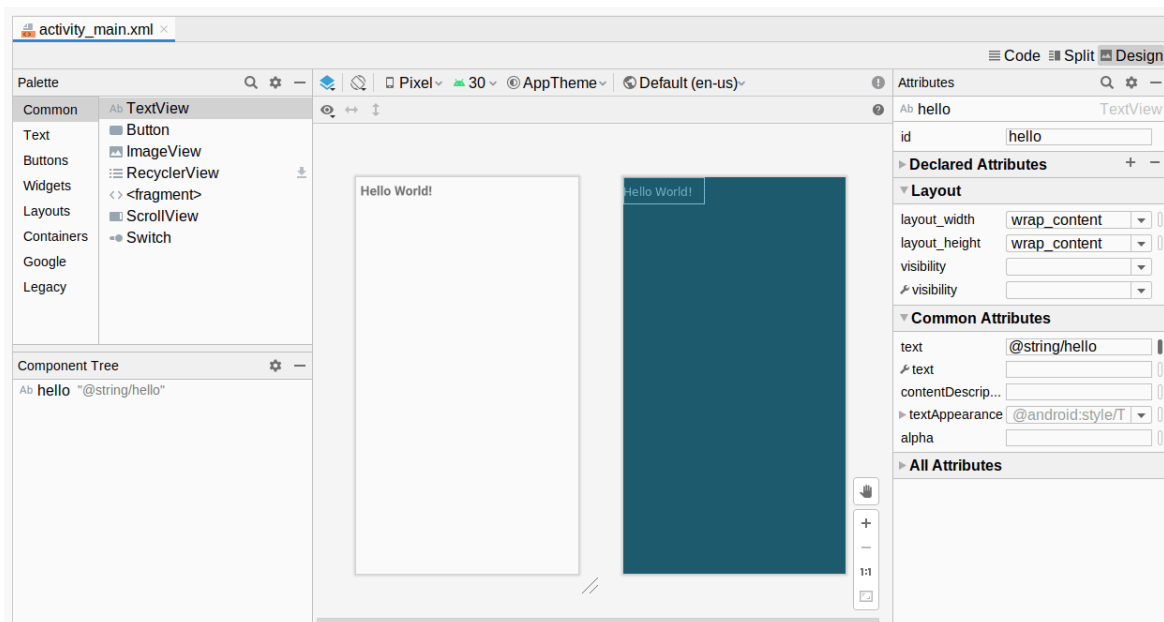


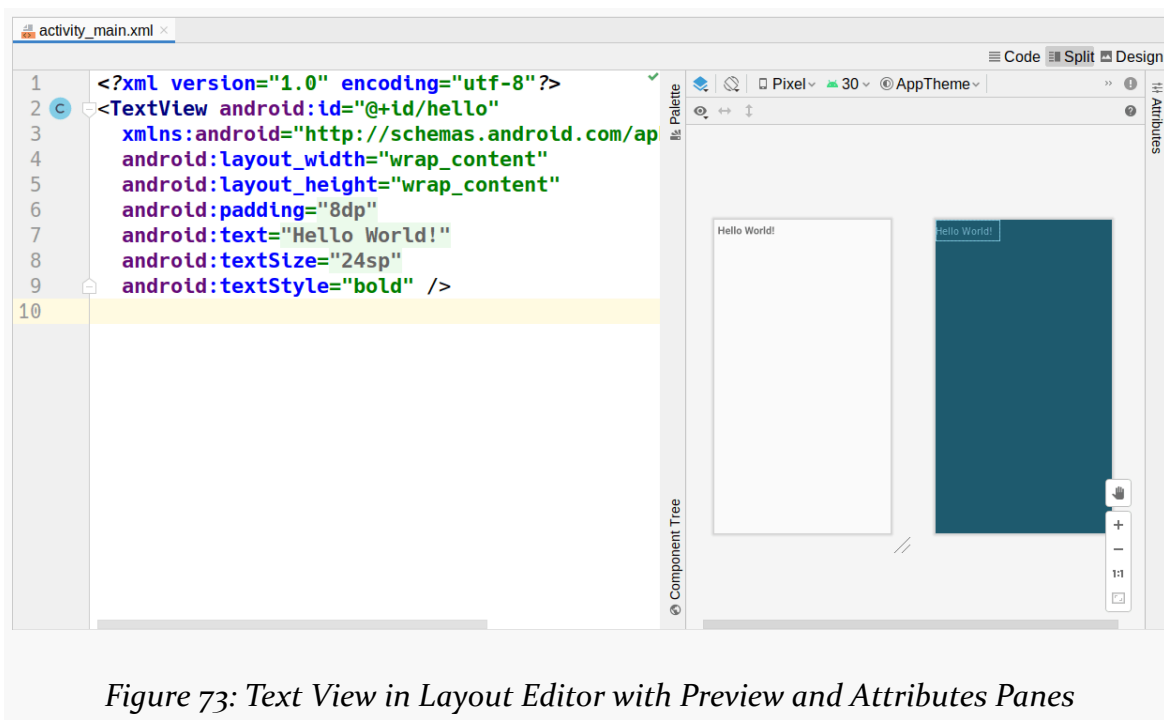
Figure 71: Android Studio Graphical Layout Editor

STARTING SIMPLE: TEXTVIEW AND BUTTON

The “Code” one allows you to edit the raw XML that is the actual content of the layout resource:



The “Split” one gives you both the text editor plus parts of the graphical layout editor:



Clicking on items in the preview will select the corresponding XML element in the text editor.

The XML editor is very useful but is also very typical: it offers basic editing with some amount of auto-completion to reduce the amount of typing that you need. If

you have done software development using other IDEs or editors, the XML editor in Android Studio should be akin to what you are used to.

The graphical editor also resembles those used in IDEs for decades. However, there has been a bit more variation in how IDEs set up those editors, and Android Studio has its own approach to this sort of tool. So, with that in mind, let's review what it offers.

Palette

The upper-left side of the graphical layout editor is the Palette tool:

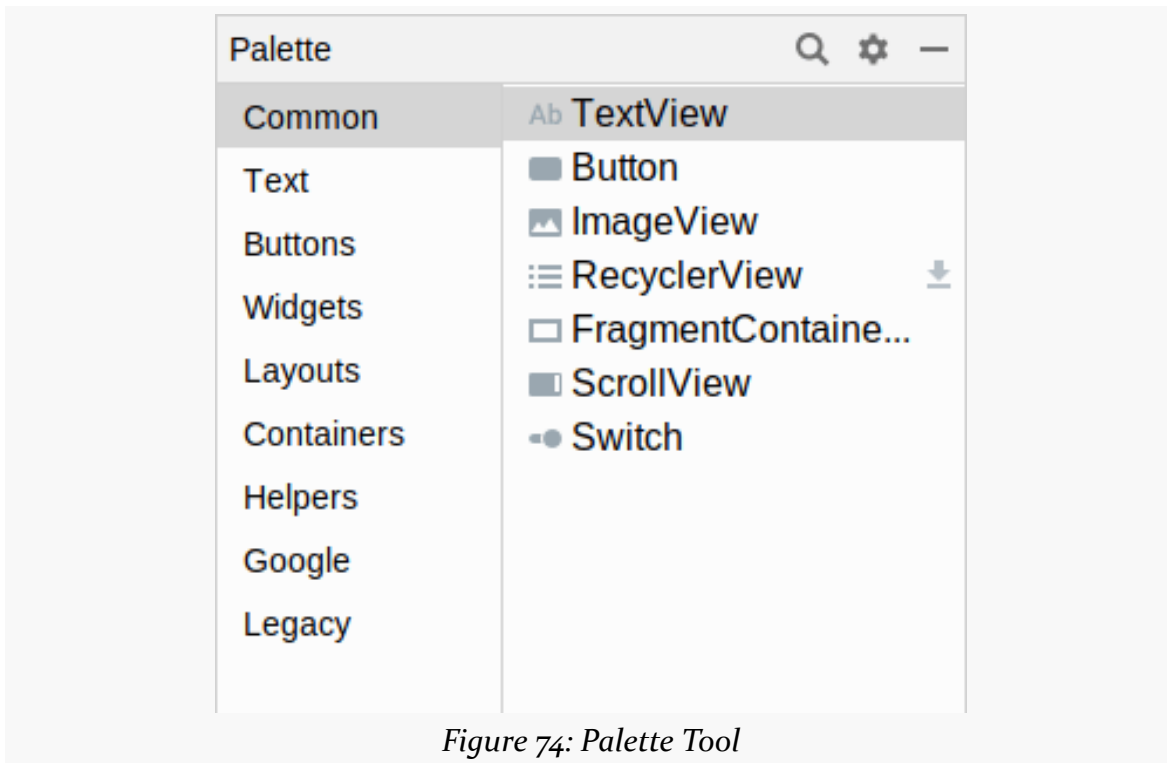


Figure 74: Palette Tool

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>`). Some — such as the `RecyclerView` shown in the above screenshot — are from libraries and may have a “download” icon adjacent to them to help illustrate that.

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 or top is a preview of what your UI should resemble, if this layout were used for the UI of an activity:



Figure 75: 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 (or below it) 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

STARTING SIMPLE: TEXTVIEW AND BUTTON

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:

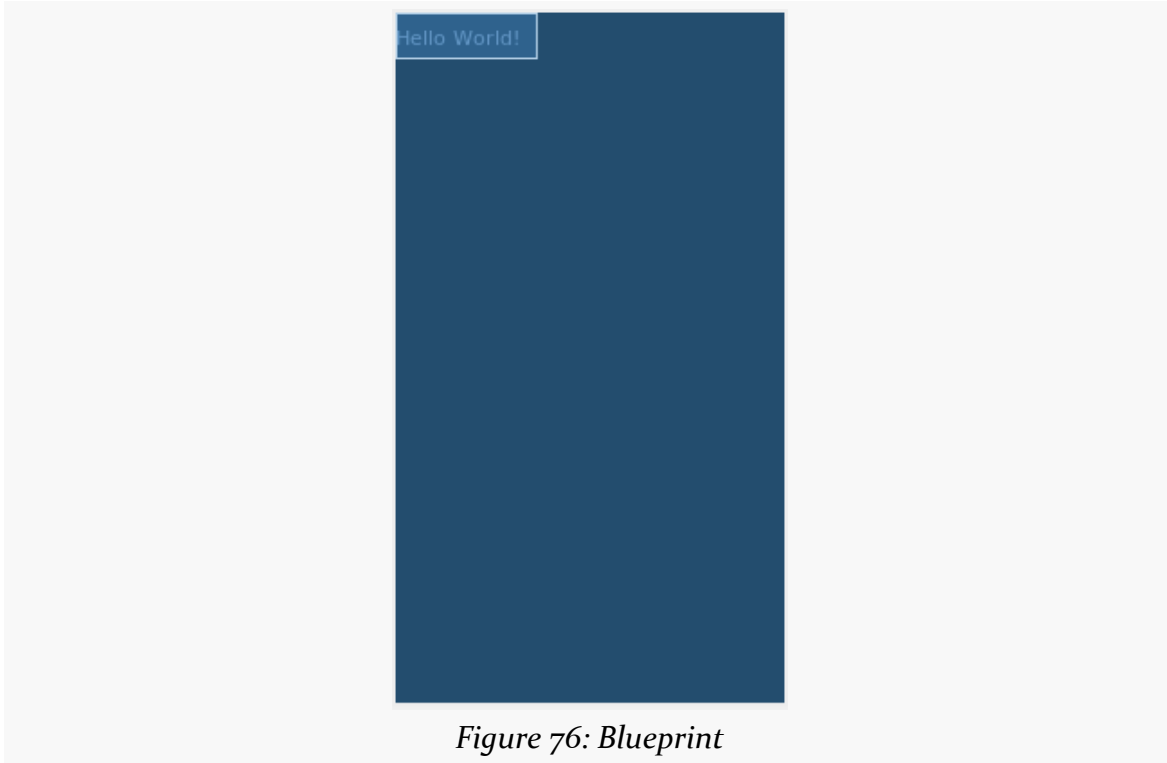


Figure 76: Blueprint

For a trivial layout resource, the blueprint view does not show you much. It will become more useful with more complex layout resources.

Preview Toolbar

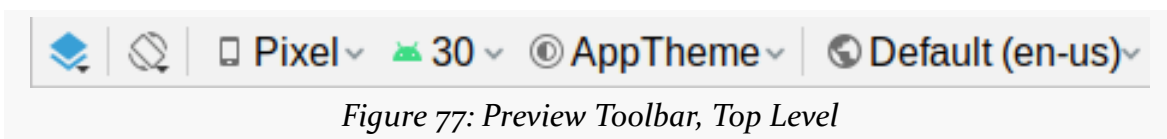


Figure 77: Preview Toolbar, Top Level

From left to right, 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

A couple of those — particularly the theme selector — pertain to topics that we will explore later in the book.

Component Tree

Towards the bottom-left corner is the component tree:

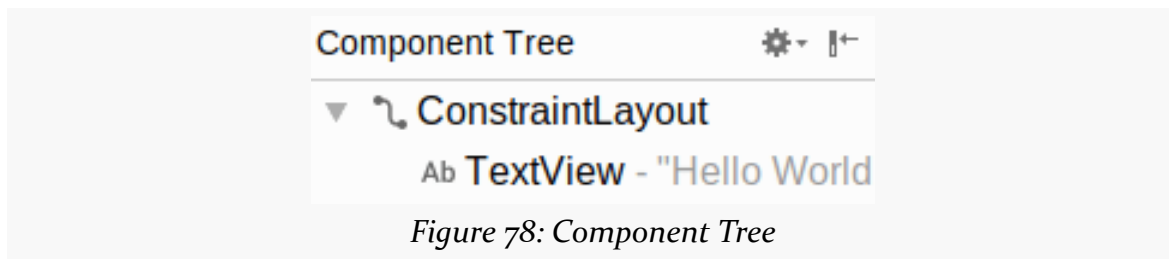


Figure 78: Component Tree

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:

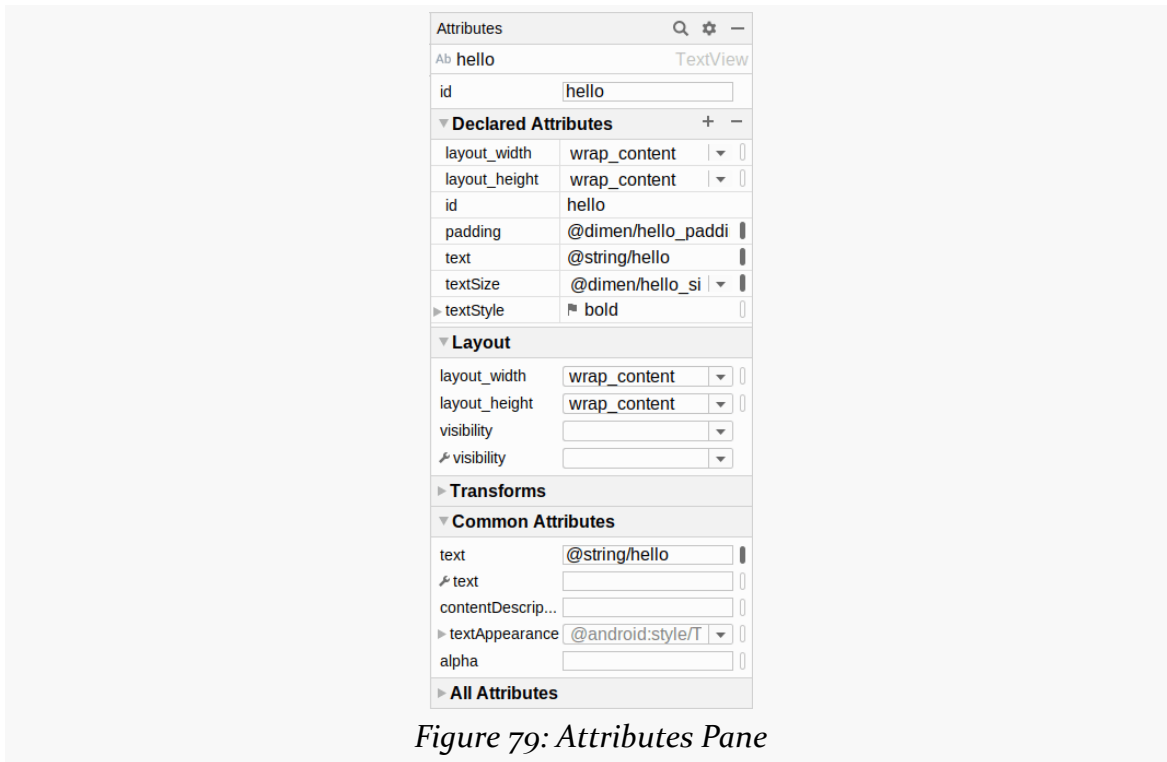


Figure 79: Attributes Pane

This is divided into sections. The “All Attributes” section, as the name suggests, lists all available attributes for this widget. The other sections highlight common subsets of the attributes. Each section can be expanded or collapsed via the triangle icon in the section header.

You can also click the magnifying glass icon in the toolbar of this pane to search for available attributes by name:

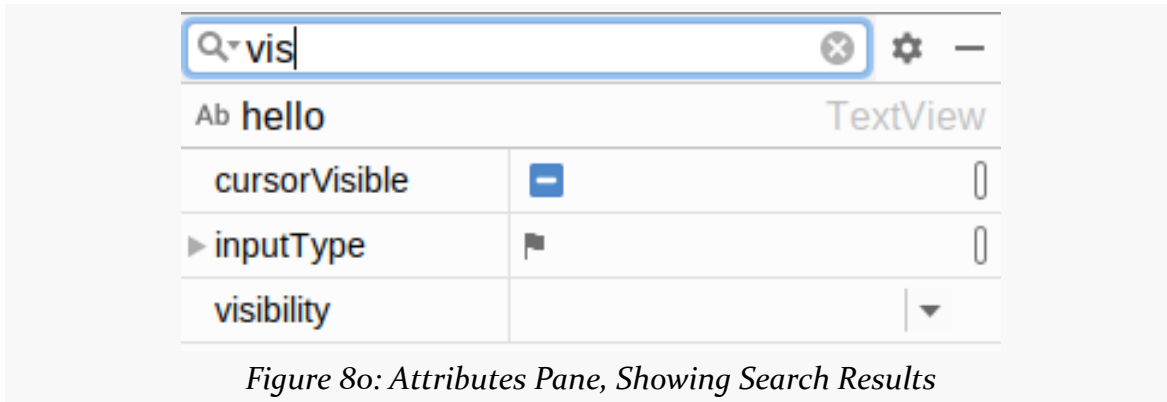


Figure 80: Attributes Pane, Showing Search Results

We will see what some of these attributes are and how to work with them over the course of the next few chapters.

TextView: Assigning Labels

Arguably, the simplest widget is the label, referred to in Android as a `TextView`. Like in most GUI toolkits, labels are bits of text that are 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) or display other text of relevance to users (e.g., messages in pop-up dialog).

As with any widget, you can create instances of `TextView` in your Java or Kotlin code by invoking a constructor, then use `setText()` methods to set the text to be displayed by the `TextView`. However, for ordinary UIs, typically you will use XML layout resources — in there, you can add a `TextView` element to the layout, with an `android:text` attribute to set the text of the label itself.

A Sample TextView

Our first sample app — `SimpleText` — is even simpler than the “Hello, World” ones from earlier in the book. The `activity_main` layout just has a `TextView` in it:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
```

STARTING SIMPLE: TEXTVIEW AND BUTTON

```
android:layout_height="wrap_content"
android:padding="@dimen/hello_padding"
android:text="@string/hello"
android:textSize="@dimen/hello_size"
android:textStyle="bold" />
```

(from [SimpleText/src/main/res/layout/activity_main.xml](#))

A TextView displays some text, set by the `android:text` attribute. Because of the `wrap_content` values for width and height, the size of this TextView will be determined by:

- The text we are putting in it (`android:text`)
- The font size of that text (`android:textSize`)
- The font style of that text (`android:textStyle`)
- The padding that we put on it (`android:padding`)

The [SamplerJ/SimpleText](#) edition of MainActivity just displays this layout, via `setContentView()`:

```
package com.commonware.jetpack.samplerj.simpletext;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

(from [SimpleText/src/main/java/com/commonware/jetpack/samplerj/simpletext/MainActivity.java](#))

...while [the Sampler/SimpleText edition](#) does the same thing, but in Kotlin instead of Java:

```
package com.commonware.jetpack.sampler.simpletext

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
}
```

```
    setContentView(R.layout.activity_main)
}
```

(from [SimpleText/src/main/java/com/commonsware/jetpack/sampler/simpletext/MainActivity.kt](https://github.com/commonsware/jetpack-sampler/blob/master/SimpleText/src/main/java/com/commonsware/jetpack/sampler/simpletext/MainActivity.kt))

Android Studio Graphical Layout Editor

The TextView widget is available in the “Common” category of the Palette in the Android Studio graphical layout editor:

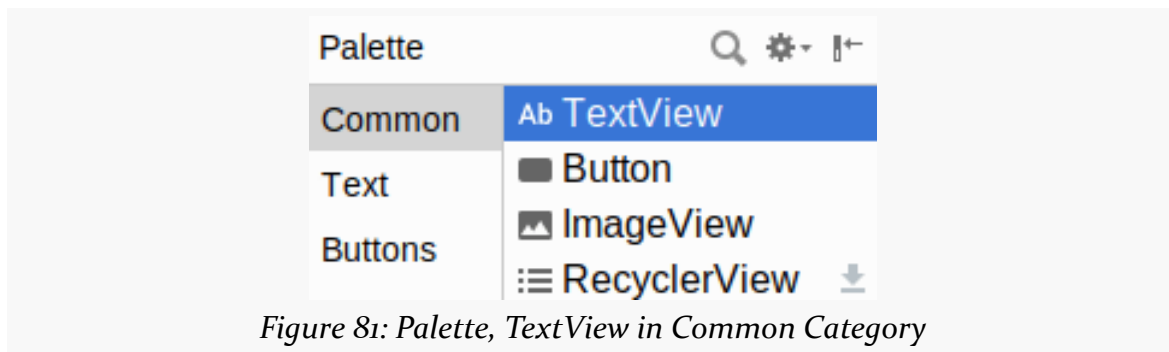


Figure 81: Palette, TextView in Common Category

(it also appears in the “Text” category)

If you want to add a TextView to a layout, just drag the 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:

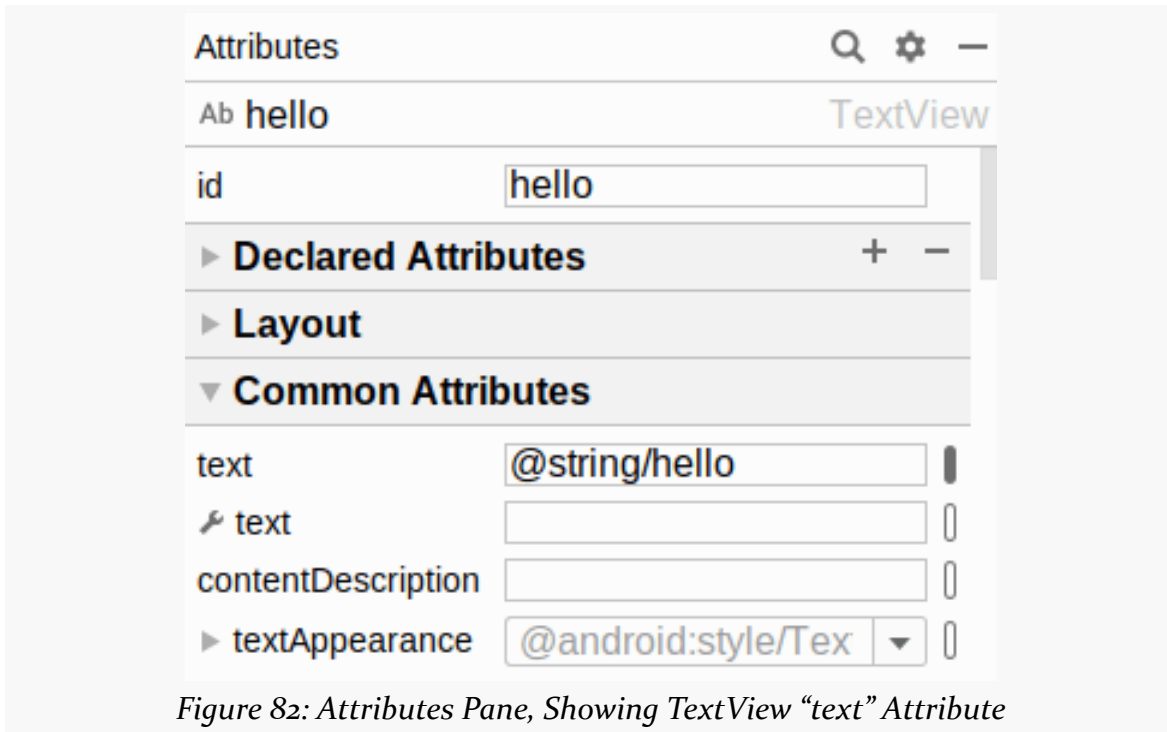


Figure 82: Attributes Pane, Showing TextView “text” Attribute

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.

STARTING SIMPLE: TEXTVIEW AND BUTTON

You can either type a literal string right in the Attributes pane row, or you can click the “O” button to the right of the field to pick a string resource:

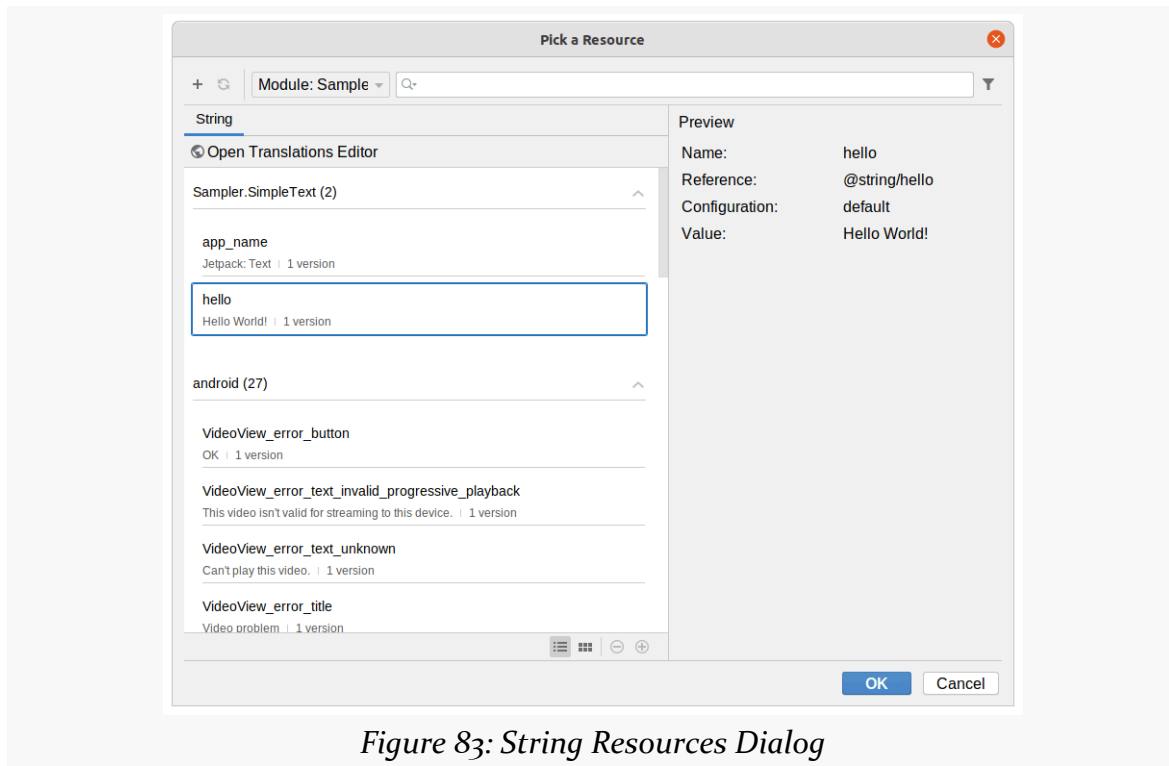


Figure 83: String Resources Dialog

STARTING SIMPLE: TEXTVIEW AND BUTTON

You can highlight one of those resources and click “OK” to use it. Or, towards the upper-left of that dialog, there is an “+” drop-down. When viewing string resources, that drop-down will offer “String Resource File” and “String Value” options. Choosing the “String Value” option will allow you to define a new string resource via another dialog:

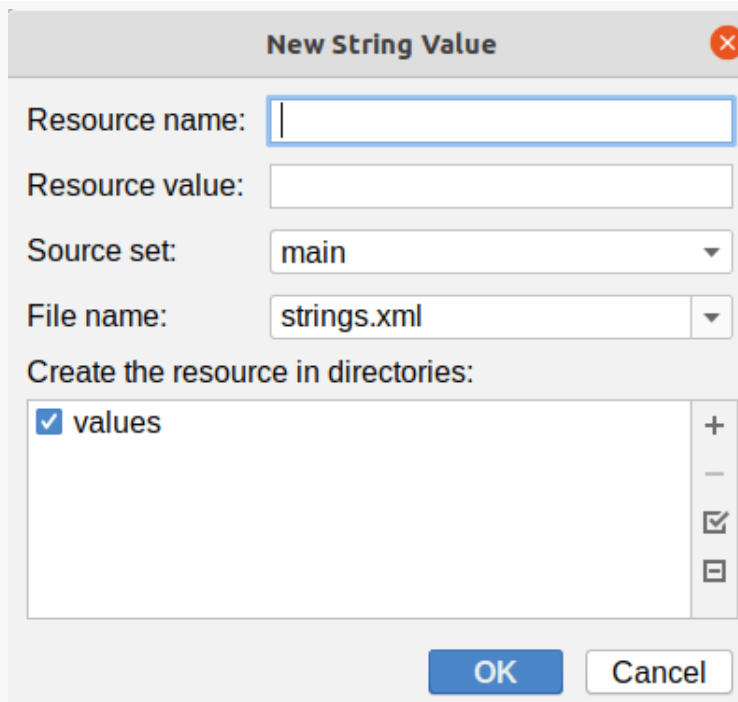


Figure 84: New String Resource Dialog

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`.

In the `SimpleText` projects, the `TextView` has an `android:text` attribute set to the `@string/hello` string resource.

Editing the ID

The “id” attribute will allow you to change the `android:id` value of the widget:



Figure 85: Attributes Pane, Showing ID Field

The value you fill in here is what goes after the `@+id/` portion (e.g., `textView2`). This works for all widgets, not just `TextView`.

In the `SimpleText` projects, the `TextView` has `hello` for its ID field contents, which results in `android:id="@+id/hello"` in the XML.

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), ARGB hex format (e.g., `#88FF0000` for a translucent red), or as a reference to a color resource (e.g., `@color/colorAccent`)

These attributes, like most others, can be modified through the Attributes pane, though many of these are in the “All Attributes” section. The `SimpleText` app sets the `textSize`, `textStyle`, and `padding` attributes.

Button: Reacting to Input

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.

A Sample Button

Our next sample app — SimpleButton — has a similar activity_main layout as the SimpleText app had:

```
<?xml version="1.0" encoding="utf-8"?>
<Button android:id="@+id/showElapsed"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="@dimen/hello_size"
    tools:text="123 seconds since started!" />
```

(from [SimpleButton/src/main/res/layout/activity_main.xml](#))

The primary difference is that this is a Button widget instead of a TextView widget. Also:

- The android:id is now showElapsed
- We skip the padding and textStyle, to adopt the standard look for buttons
- We skip the android:text that normally would be here, as we are going to provide the text at runtime via our Java or Kotlin code
- We have a tools:text attribute... which raises a question

Hey, What Is That tools: Thing?

You will notice that our Button element has a tools:text attribute.

Attributes in the tools: namespace are suggestions to the development tools and have no impact on the behavior of your app when it runs. They are here to help make Android Studio work a bit better, particularly with respect to the graphical layout editor.

Normally, we have an android:text attribute on TextView and subclasses, and that provides the text. Here, though, we skip that attribute, as we are going to provide the text at runtime. However, that makes the graphical layout editor less useful:

- The caption of the Button shows up blank
- Since the width and height of the Button are each wrap_content, the graphical layout editor does not know how big to make the Button

tools:text says, “hey, Android Studio! use this for the Button caption for the

graphical layout editor!”. This value will not show up when you run the app; it will only appear in the IDE. Typically, you set `tools:text` to either:

- A reasonable value, to see what that looks like
- Some extreme value, such as a really long string, to see what that looks like, in case you find that you need to adjust the GUI design to deal with that

Android Studio Graphical Layout Editor

The Button widget is available in the “Buttons” portion of the Palette in the Android Studio graphical layout editor:

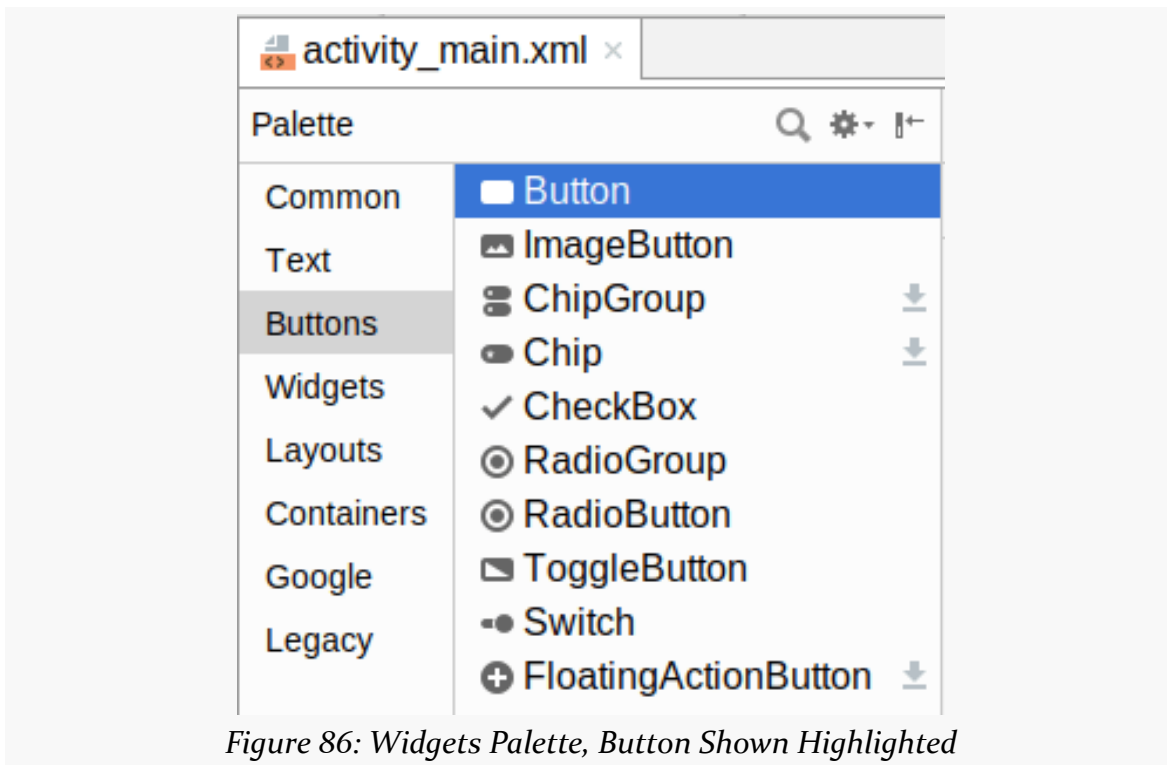


Figure 86: Widgets Palette, Button Shown Highlighted

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 Using View Binding

In the SimpleButton sample, we want to show the time since the activity was

STARTING SIMPLE: TEXTVIEW AND BUTTON

displayed in the button caption. Tapping the button should update the caption to show the now-current elapsed time.

This implies that:

- We can find out when the button is clicked, and
- We can set the `android:text` attribute at runtime with a generated value

Updating the caption of the Button is a matter of calling `setText()` on the Button with the desired caption. However, there are a few approaches for doing that, as well as for finding out about the button clicks. The recommended current approach for getting access to the Button widget, for general-purpose use, is to use “view binding”.

With view binding, the Android build tools code-generate a Java class for you, based on each one of your layout resources. That class not only helps you set up the layout, but it gives you fields for accessing each of the named widgets within that layout.

Both [the SamplerJ/SimpleButton Java edition](#) and [the Sampler/SimpleButton Kotlin edition](#) of this sample use view binding.

View binding is enabled as an option via a new closure in the android closure in your module’s `build.gradle` file:

```
buildFeatures {  
    viewBinding true  
}
```

(from [SimpleButton/build.gradle](#))

`buildFeatures` does “pretty much what it says on the tin”: it enables optional build features that get added to our Android builds. Here, we are opting into view binding via `viewBinding true`.

NOTE: You will see some sample Gradle files with `viewBinding = true`. That syntax worked on older versions of Android Studio but was replaced by `viewBinding true` in Arctic Fox.

Adding those lines automatically sets up view binding for each of our layouts. Right now, we just have a single layout: `activity_main`.

The class name for the code-generated class is derived from the layout name, where `names_like_this` get converted into `NamesLikeThis` and have `Binding` appended. So,

STARTING SIMPLE: TEXTVIEW AND BUTTON

since our layout resource is `activity_main.xml`, we get `ActivityMainBinding`. 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.commonware.jetpack.sampler.simplebutton.databinding.ActivityMainBinding
```

(if you are wondering why this is “view binding”, but the package name has databinding, that is for historical reasons — just roll with it)

Our activity can then reference `ActivityMainBinding` and use it to set up the UI and get a reference to the `showElapsed` widget, in Java:

```
package com.commonware.jetpack.samplerj.simplebutton;

import android.os.Bundle;
import android.os.SystemClock;
import android.widget.Button;
import com.commonware.jetpack.samplerj.simplebutton.databinding.ActivityMainBinding;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    private final long startTimeMs = SystemClock.elapsedRealtime();
    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
        binding.showElapsed.setOnClickListener(v -> updateButton());
        updateButton();
    }

    void updateButton() {
        long nowMs = SystemClock.elapsedRealtime();
        String caption = getString(R.string.elapsed, (nowMs - startTimeMs) / 1000);

        binding.showElapsed.setText(caption);
    }
}
```

(from [SimpleButton/src/main/java/com/commonware/jetpack/samplerj/simplebutton/MainActivity.java](#))

...and Kotlin:

```
package com.commonware.jetpack.sampler.simplebutton

import android.os.Bundle
import android.os.SystemClock
import androidx.appcompat.app.AppCompatActivity
import com.commonware.jetpack.sampler.simplebutton.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
```

STARTING SIMPLE: TEXTVIEW AND BUTTON

```
private val startTimeMs = SystemClock.elapsedRealtime()
private lateinit var binding: ActivityMainBinding

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    binding.showElapsed.setOnClickListener { updateButton() }
    updateButton()
}

private fun updateButton() {
    val nowMs = SystemClock.elapsedRealtime()
    val caption = getString(R.string.elapsed, (nowMs - startTimeMs) / 1000)

    binding.showElapsed.text = caption
}
}
```

(from [SimpleButton/src/main/java/com/commonsware/jetpack/sampler/simplebutton/MainActivity.kt](#))

We get an `ActivityMainBinding` instance by calling the static `inflate()` method on it, passing in a `LayoutInflater` that we get from `getLayoutInflater()`. A `LayoutInflater` knows how to take a layout resource and create the corresponding Java objects representing all of the widgets and containers in that resource. When we called `setContentView(R.layout.activity_main)` before, under the covers, `setContentView()` used a `LayoutInflater`. `ActivityMainBinding` also uses a `LayoutInflater`.

Given that `ActivityMainBinding` object, we can call `setContentView()`, this time not passing in `R.layout.activity_main`. We already inflated the layout using `ActivityMainBinding` and `LayoutInflater` — we do not need to do it twice. Instead, we call `getRoot()` on our binding object, which represents the root of our layout view hierarchy. Passing that to `setContentView()` sets up our UI, just as `setContentView(R.layout.activity_main)` did.

But the binding object *also* gives us access to our widgets. Specifically, `ActivityMainBinding` has a `showElapsed` field, named after the `android:id` that we used for the `<Button>` in the layout. `showElapsed` is a Java `Button` object representing this widget, and we can do things like call `setOnClickListener()` to arrange to get control when the user clicks the button.

Natively, `setOnClickListener()` takes an `OnClickListener` callback object. In both the Java and the Kotlin examples, we call `setOnClickListener()` on the `Button`, we are using a lambda expression that Java or Kotlin will convert into an `OnClickListener` for us, with the body of the lambda expression forming the body of the `onClick()` method and being called when the user clicks the button. There,

STARTING SIMPLE: TEXTVIEW AND BUTTON

we call an `updateButton()` method, which we also call from `onCreate()`.

`updateButton()` uses `SystemClock.elapsedRealtime()` to get the number of milliseconds that have elapsed since the device booted. `SystemClock` is a class supplied by the Android SDK that returns various time values that are tied to device activity. We use that same `elapsedRealtime()` method to populate a `startTimeMs` field in the activity, which will record when the activity was displayed. Hence, the number of milliseconds between the activity being displayed and now is a matter of subtracting `startTimeMs` from `nowMs`.

To display that in the Button, `updateButton()` calls `getString()` on the activity. `getString()` returns the value of a string resource, given the ID of that string resource. Here, we are looking to pull in an elapsed string resource. The `SimpleButton` project defines that using a combination of a message and a placeholder:

```
<resources>
  <string name="app_name">Jetpack: Button</string>
  <string name="elapsed">%d seconds since started!</string>
</resources>
```

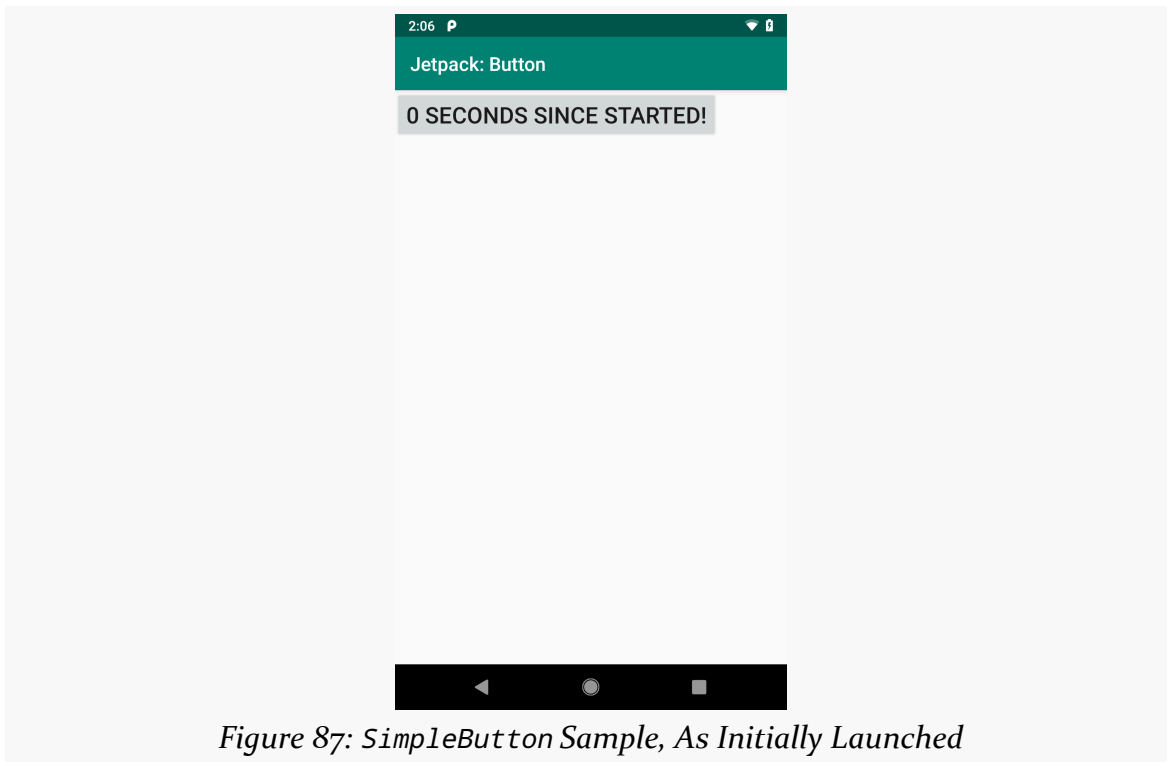
(from [SimpleButton/src/main/res/values/strings.xml](#))

String resources support the same placeholder patterns as does Java's `String.format()` method, which is largely the same as what you might use in `sprintf()` in C/C++ development. Here, `%d` says “we will supply an integer to fill in here”, which we do via the second parameter to `getString()`. The result is that caption will hold the string resource with the `%d` replaced by the number of seconds since the activity was started.

That caption value is then passed to `setText()` on our Button, which causes it to show the caption.

STARTING SIMPLE: TEXTVIEW AND BUTTON

Initially, that caption will show that 0 seconds have elapsed:



STARTING SIMPLE: TEXTVIEW AND BUTTON

If you tap it later, it will show the number of seconds since the activity was started:



Figure 88: SimpleButton Sample, A Little While Later

Note that the Button changes size as the caption changes length. That is because the Button uses `wrap_content` for its width, so as the content changes, so will the Button width.

So, we:

- Use the code-generated binding class to set up our UI based on our layout resource
- Used the `showElapsed` field on that binding class to get at our Button object
- Used `setOnClickListener()` and `setText()` on the Button to find out when the button is clicked and update its caption, respectively

There are alternatives to view binding for doing all of this — we will discuss those more [later in the book](#).

The Curious Case of the Missing R

The Java and Kotlin code both refer to these R values. The R class gets code generated by the IDE and build tools based on our current roster of resources. Each resource gets a corresponding R value based on the type of resource (`R.layout`, `R.id`, `R.string`, etc.) and the name of the resource (`R.layout.activity_main`, `R.id.showElapsed`, `R.string.elapsed`, etc.).

Sometimes, when you are working on your Java or Kotlin code, you will run into cases where you cannot seem to use the R class to reference your resources. There are two main scenarios:

- The IDE says that it does not know what R is
- The IDE says that it knows what R is but does not recognize a particular resource that you are trying to reference (e.g., you have `R.layout.foo`, and either `layout` or `foo` are reported as not being recognized)

These problems come about due to where that R class lives and when it is created (or updated).

Where R Lives

All of our Java and Kotlin classes reside in some “package”. In the starter project, that package is `com.commonware.helloworld`. Our one-and-only class, `MainActivity`, is in that package. The samples shown in this chapter are in different packages, such as `com.commonware.jetpack.sampler.simplebutton` for the Kotlin edition of the `SimpleButton` sample.

The R class will be code-generated by the build tools in the Java package identified by the package attribute in the `<manifest>` element. So, for a project whose package is `com.commonware.jetpack.sampler.simplebutton`, the fully-qualified class name for R is `com.commonware.jetpack.sampler.simplebutton.R`.

Java and Kotlin share a common rule: you do not need to import classes that are in the same package as the class that you are in. So, in our projects, `MainActivity` can refer to R without an import statement, as both `MainActivity` and R are in the project’s package (e.g., `com.commonware.jetpack.sampler.simplebutton`).

If you try referencing R, and Android Studio complains that it does not know what R is, consider whether your class is in a different package than the one defined in the

`<manifest>` element. If it is, you will need to add an `import` statement to pull in your `R` class.

When R Is Created

`R` is created as part of building your project. To allow code-completion and other assistance features of Android Studio to work, `R` gets re-created whenever you add, rename, or remove a resource, as `R` is an “index” of all of the resources in your app. However, occasionally, you will use some feature of Android Studio that happens to add, rename, or remove a resource, but Android Studio fails to regenerate that `R` class.

If Android Studio knows what `R` is but does not recognize some particular resource, try manually building the project, via **Build > Rebuild Project** from the Android Studio main menu. This forces `R` to get regenerated, and it may clear up your problem.

When R Is Not Created

Sometimes, though, Android Studio does not know what `R` is because `R` is missing entirely.

If there is some bug in one of your resources — or, as it turns out, if there is some bug in your manifest — `R` may not be regenerated. If Android Studio says that it does not know what `R` is, and either you already have the `import` statement or you should not need one (since `R` and your code are in the same package), then it is likely that there is some problem in a resource or your manifest.

If you manually build the project — again, via **Build > Rebuild Project** from the Android Studio main menu — you should get a build error telling you exactly what resource is flawed, so you can try to fix it.

Package Names

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.commonware.myapplication`), where you own the domain in question. That way, it is unlikely that anyone else will accidentally collide with the same name.

Debugging Your App

Now that we are starting to manipulate widgets, the odds increase that we are going to somehow do it wrong, and our app will crash. Usually, when your app crashes, the OS will show a dialog box:

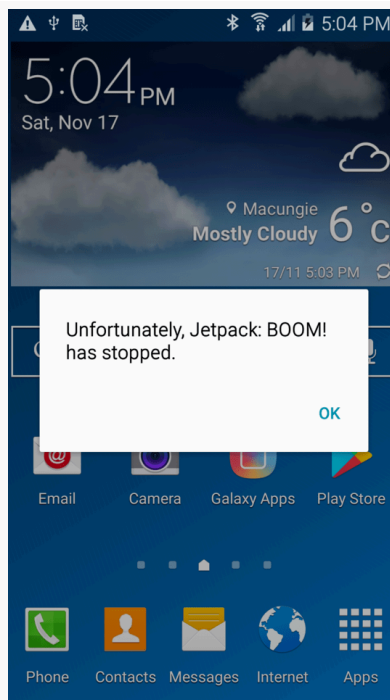


Figure 89: Android 5.0 App Crash Dialog

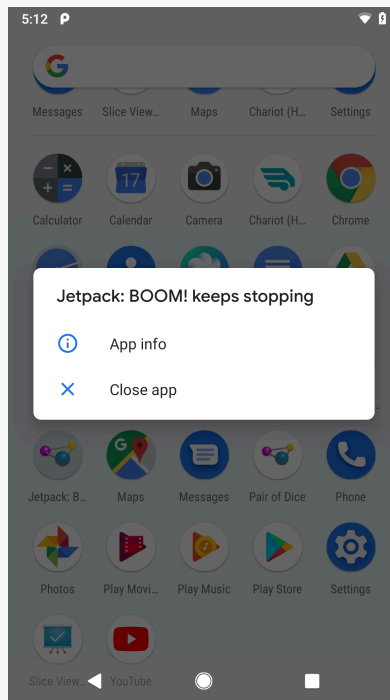


Figure 90: Android 9.0 Dialog After Repeated Crashes

In this chapter, we will cover a few tips on how to debug these sorts of issues.

Get Thee To a Stack Trace

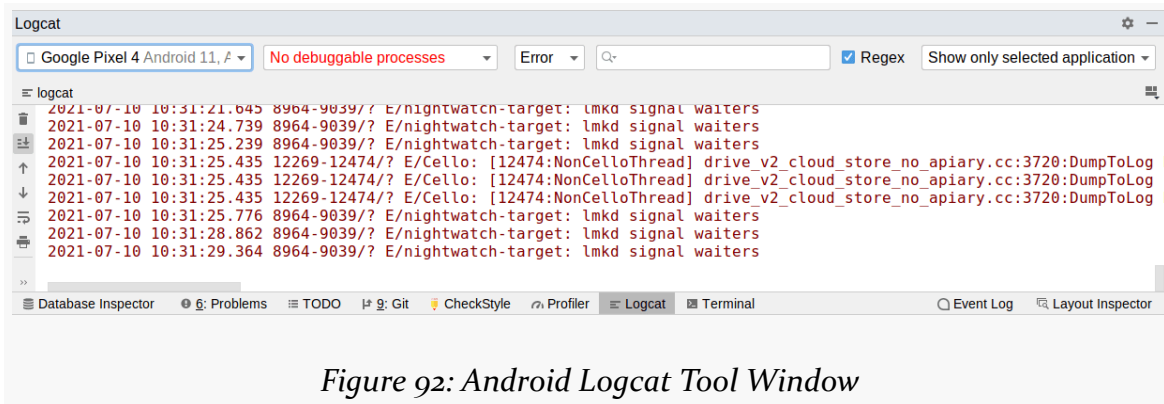
If it seems like your app has crashed, the first thing you will want to do is examine the stack trace that is associated with this crash. These are logged to a facility known as Logcat, on your device or emulator. You can view those logs using the Logcat tool in Android Studio.

DEBUGGING YOUR APP

The Logcat tool 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:



Clicking on that will bring up some Android-specific logs in an Logcat tool window:



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.

The toolbar across the top of the Logcat window has a drop-down list of available devices and running emulators. Whichever one is selected there is the source of the log messages seen in the main area of the Logcat tool window.

If your app crashes, most of the time, there will be an associated Java stack trace (even if you are writing Kotlin code). For example the stack trace that triggered the crash dialog shown at the start of this chapter is:

DEBUGGING YOUR APP

```
E/AndroidRuntime: FATAL EXCEPTION: main
Process: com.commonware.jetpack.sampler.simpleboom, PID: 14064
java.lang.RuntimeException: Unable to start activity...
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2693)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2758)
    at android.app.ActivityThread.access$900(ActivityThread.java:177)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1448)
    at android.os.Handler.dispatchMessage(Handler.java:102)
    at android.os.Looper.loop(Looper.java:145)
    at android.app.ActivityThread.main(ActivityThread.java:5942)
    at java.lang.reflect.Method.invoke(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:372)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run...
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1195)
Caused by: android.content.res.Resources$NotFoundException: String resource...
    at android.content.res.Resources.getText(Resources.java:1334)
    at android.widget.TextView.setText(TextView.java:4917)
    at com.commonware.jetpack.sampler.simpleboom.MainActivity...
    at com.commonware.jetpack.sampler.simpleboom.MainActivity...
    at android.app.Activity.performCreate(Activity.java:6283)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1119)
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2646)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2758)
    at android.app.ActivityThread.access$900(ActivityThread.java:177)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1448)
    at android.os.Handler.dispatchMessage(Handler.java:102)
    at android.os.Looper.loop(Looper.java:145)
    at android.app.ActivityThread.main(ActivityThread.java:5942)
    at java.lang.reflect.Method.invoke(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:372)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run...
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1195)
```

(note: some lines were truncated with ... to try to make the output fit the page width better)

Most of the time — though not always — you should be able to find references to your code in the stack trace, to know where you were when you crashed.

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 (e.g., only show “Error” severity)
- 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 (e.g., “No Filters”)

Running Your App in the Debugger

The SimpleBoom module in the [Sampler](#) and [Sampler.J](#) projects is very similar to the SimpleButton example we looked at in [the chapter on widgets](#). There are three main differences:

- The button now has a text caption defined in the layout via the `android:text` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<Button android:id="@+id/showElapsed"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/elapsed_caption"
    android:textSize="@dimen/hello_size" />
```

(from [SimpleBoom/src/main/res/layout/activity_main.xml](#))

DEBUGGING YOUR APP

- We no longer immediately update the button when starting the activity, so that caption shows up:

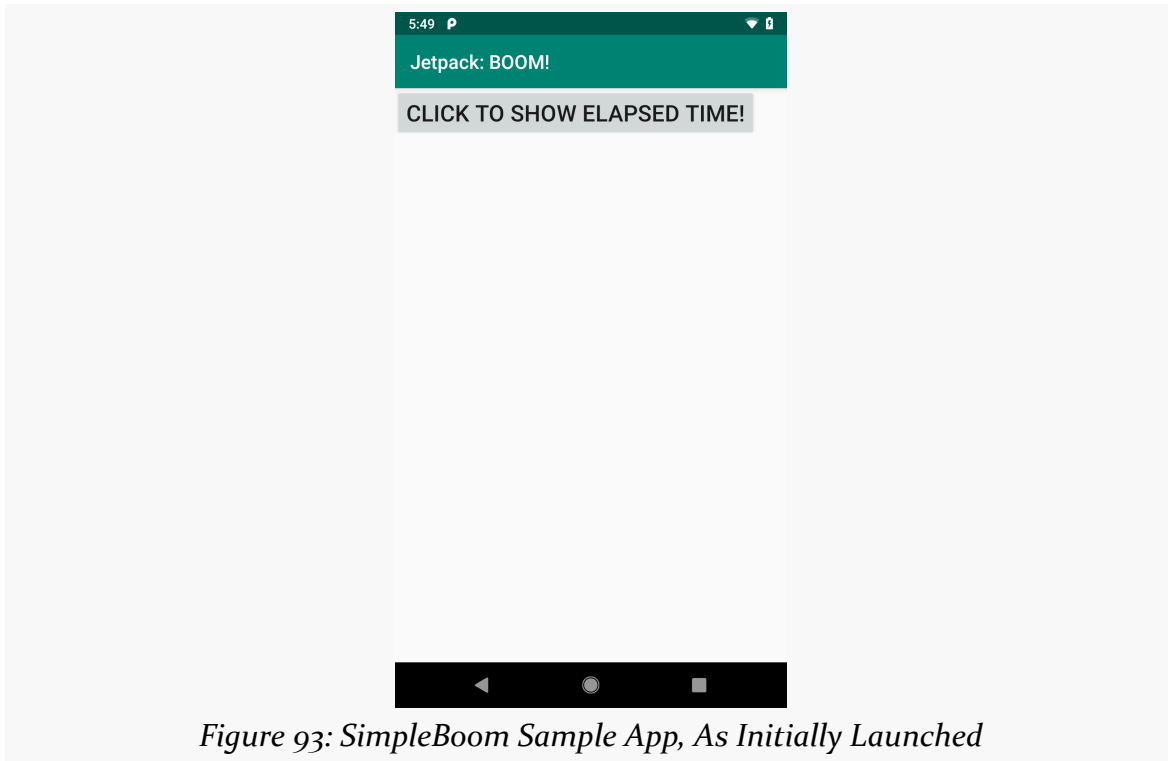


Figure 93: SimpleBoom Sample App, As Initially Launched

- We crash when the user taps the button

Sometimes, the stack trace alone will give you enough information to know how to fix the bug. If not, you can run your app in the debugger to see exactly how your code is executing and perhaps get a better sense for your mistake.

Setting Breakpoints

In our Java and Kotlin code editors in Android Studio, the vertical bar on the left is called “the gutter”:



Figure 94: Editor, with Gutter Area Highlighted

The author of this book likes having the line numbers visible. To control whether those show up, go to Files > Settings > Editor > General > Appearance — the “Show line numbers” checkbox controls whether the gutter shows line numbers.

Clicking in the gutter on a line of source code will set a breakpoint, which appears as a red dot in the gutter and a pink highlight on the line:

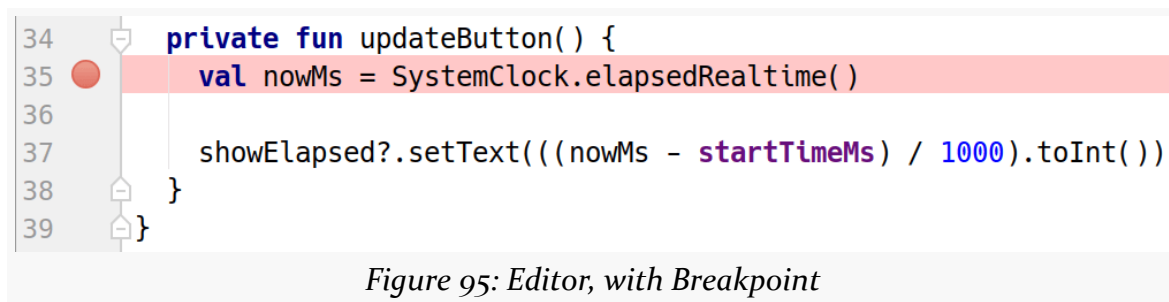


Figure 95: Editor, with Breakpoint

DEBUGGING YOUR APP

When you run your app using the debugger, when code execution reaches lines with a breakpoint, execution stops and you can use the IDE to learn more about what is going on.

For lines containing lambda expressions, when you click in the gutter to set a breakpoint, you will get a drop-down menu of options for what the breakpoint should affect:

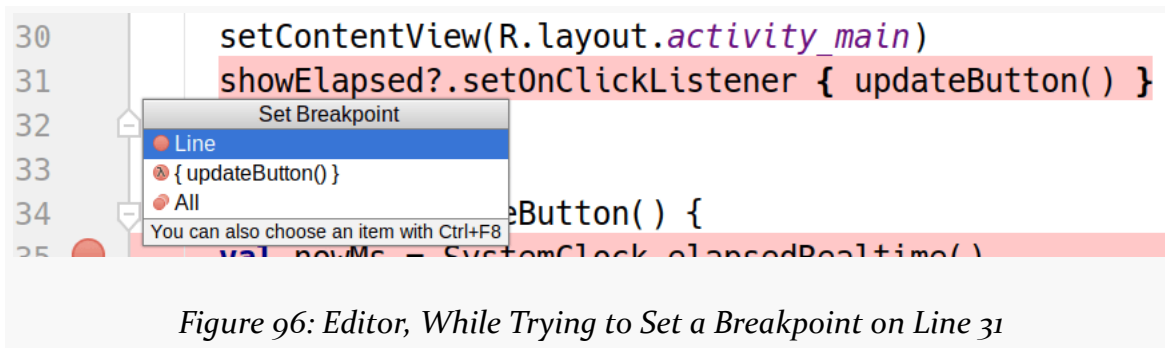


Figure 96: Editor, While Trying to Set a Breakpoint on Line 31

Your options are:

- Set the breakpoint for the initial code on the line (“Line”)
- Set the breakpoint to trigger when the lambda expression is executed (“{ updateButton() }”)
- Both (“All”)

Launching the Debugger

To run the app and have the breakpoints take effect, you need to run it in the debugger. You can do this from the Run > Debug main menu option or the “green bug” toolbar icon:

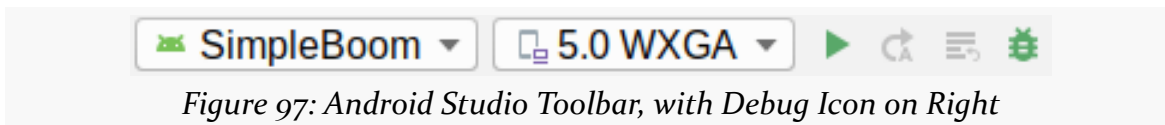


Figure 97: Android Studio Toolbar, with Debug Icon on Right

DEBUGGING YOUR APP

This behaves a lot like when you normally run the app. However, the app will run more slowly, and when the breakpoint is reached, your IDE window will open a Debug tool that shows you what is going on:

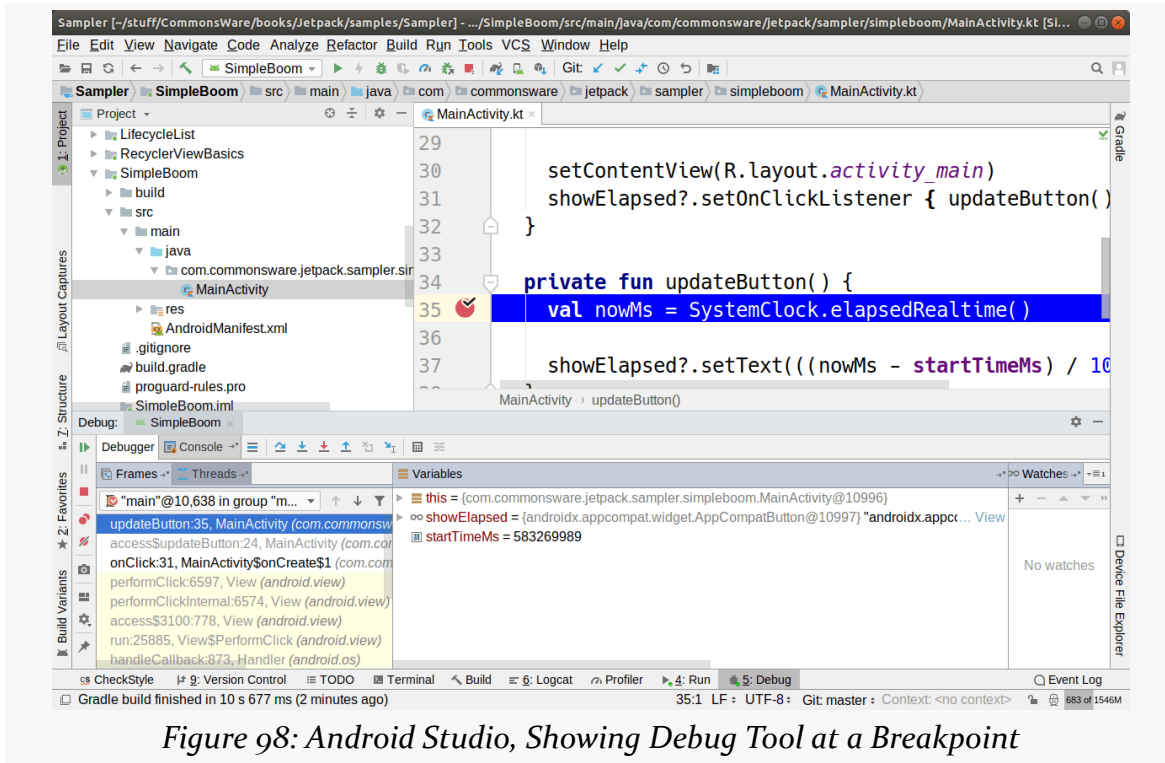


Figure 98: Android Studio, Showing Debug Tool at a Breakpoint

Examining Objects

The middle of the Debug tool shows a “Variables” list. This will contain local variables, fields/properties of the current object, and other values that your code might be referencing:

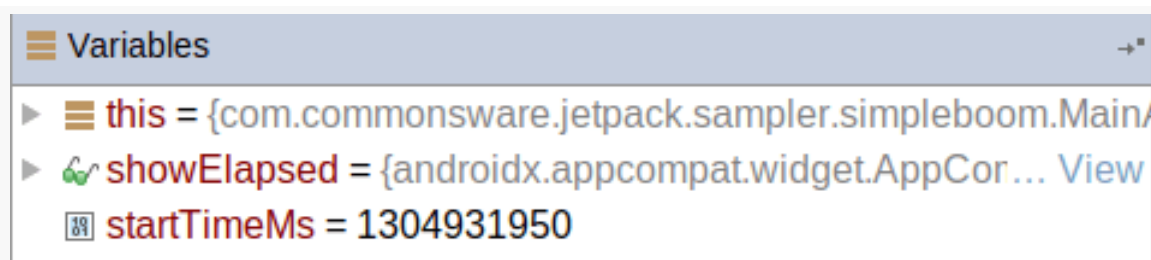


Figure 99: Variables Pane of Debug Tool

Simple types, like the Long value nowMs, just show their value. More complex types will show you their type and can be explored in turn using the gray triangle to expand the object tree:

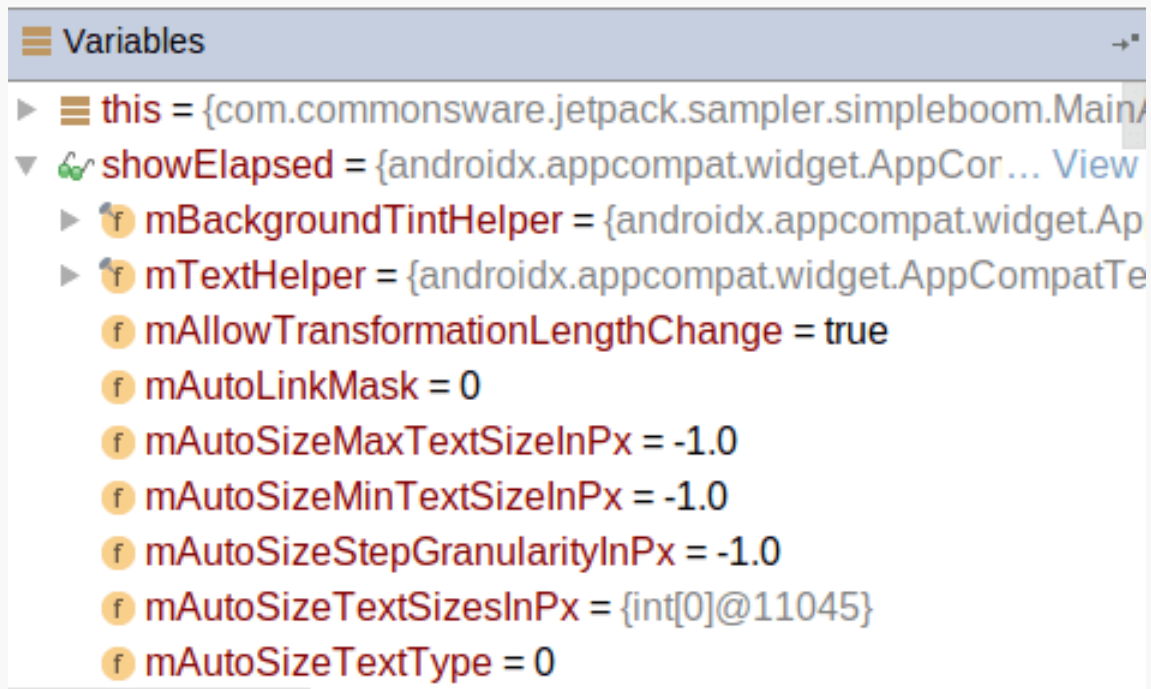


Figure 100: Variables Pane of Debug Tool, Showing Object Contents

This can help you understand the data at this point in the code.

Stepping Through the Code

The toolbar towards the top of the Debug tool has a few buttons that let you step through the code to follow along as it gets executed:

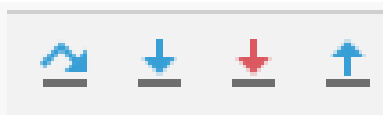


Figure 101: Code Step Toolbar Buttons in Debug Tool

From left to right, these are:

- Execute this statement and move into the next one

DEBUGGING YOUR APP

- Step into the function being called in this statement, if it is not an Android SDK function
- Step into the function being called in this statement, even if it is an Android SDK function
- Execute far enough to exit the function that we are in and then stop

Also, there is a tool strip on the left with a few useful buttons:

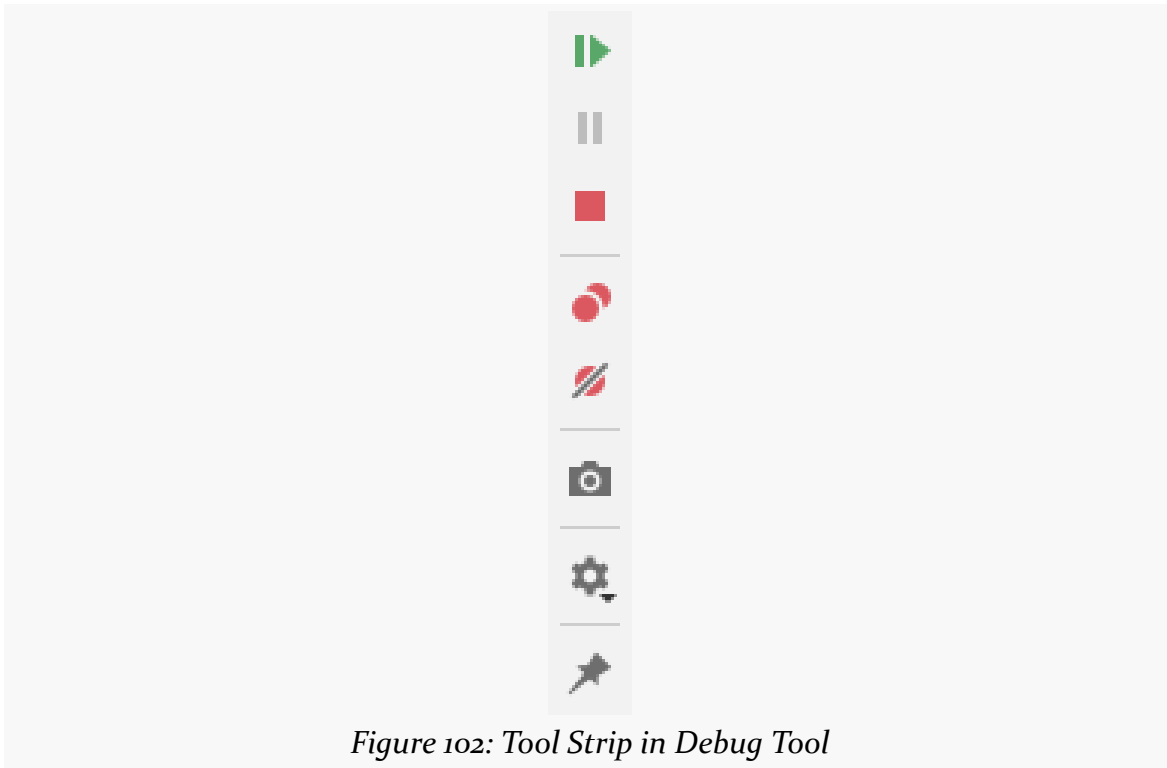


Figure 102: Tool Strip in Debug Tool

Of particular note:

- The top-most button (green triangle) will start executing code until the next breakpoint or crash
- The red square terminates your process
- The fourth button (two overlapping red dots) brings up a dialog box showing you all of your breakpoints

So, Where Did We Go Wrong?

In both the Java and Kotlin editions of MainActivity and its updateButton()

DEBUGGING YOUR APP

method, we have a simpler bit of code that tries to set the caption to just be the number of seconds that have elapsed:

```
private fun updateButton() {  
    val nowMs = SystemClock.elapsedRealtime()  
  
    binding.showElapsed.setText(((nowMs - startTimeMs) / 1000).toInt())  
}
```

(from [SimpleBoom/src/main/java/com/commonsware/jetpack/sampler/simpleboom/MainActivity.kt](#))

```
void updateButton() {  
    long nowMs = SystemClock.elapsedRealtime();  
    int seconds = (int)((nowMs - startTimeMs) / 1000);  
  
    binding.showElapsed.setText(seconds);  
}
```

(from [SimpleBoom/src/main/java/com/commonsware/jetpack/samplerj/simpleboom/MainActivity.java](#))

This code compiles, because `setText()` on a `Button` has a variant that accepts an `Int` as a parameter. We are passing an `Int` that represents the number of seconds.

Let's look at the relevant portion of the stack trace again:

```
Caused by: android.content.res.Resources$NotFoundException: String resource ID #0x11  
    at android.content.res.Resources.getText(Resources.java:1334)  
    at android.widget.TextView.setText(TextView.java:4917)  
    at com.commonsware.jetpack.sampler.simpleboom.MainActivity.updateButton...
```

As it turns out, `setText(Int)` does not show that `Int` value as text in the `Button` caption. Instead, Android is expecting that to be a reference to a string resource, such as `R.string.elapsed`. Those `R` values are all integers, and so many functions in the Android SDK that accept an `Int` are expecting a resource ID, not some arbitrary value as we are providing. In particular, if you get crash with an `android.content.res.Resources$NotFoundException`, there is a good chance that you are passing in a value to a function that is not a valid resource ID.

The correct way to implement this would be to convert the numeric value to a `String`, then pass that to `setText()`:

DEBUGGING YOUR APP

```
private fun updateButton() {  
    val nowMs = SystemClock.elapsedRealtime()  
  
    binding.showElapsed?.setText(((nowMs - startTimeMs) / 1000).toString())  
}
```

```
void updateButton() {  
    long nowMs = SystemClock.elapsedRealtime();  
    int seconds = (int)((nowMs - startTimeMs) / 1000);  
  
    binding.showElapsed.setText(Integer.toString(seconds));  
}
```


Introducing ConstraintLayout

The starter app that we examined in the opening chapters had a layout like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

This wraps a single `TextView` in a `ConstraintLayout`. For the purposes of what this “hello, world!” app does, the `ConstraintLayout` is pointless — we saw activities using just a `TextView` in [the chapter on widgets](#).

However, more often than not, you will have more than one widget in your layout resources. When you do, you will always have some sort of container, and frequently that container will be a `ConstraintLayout`.

In this chapter, we will review what containers are and why we use them, examine

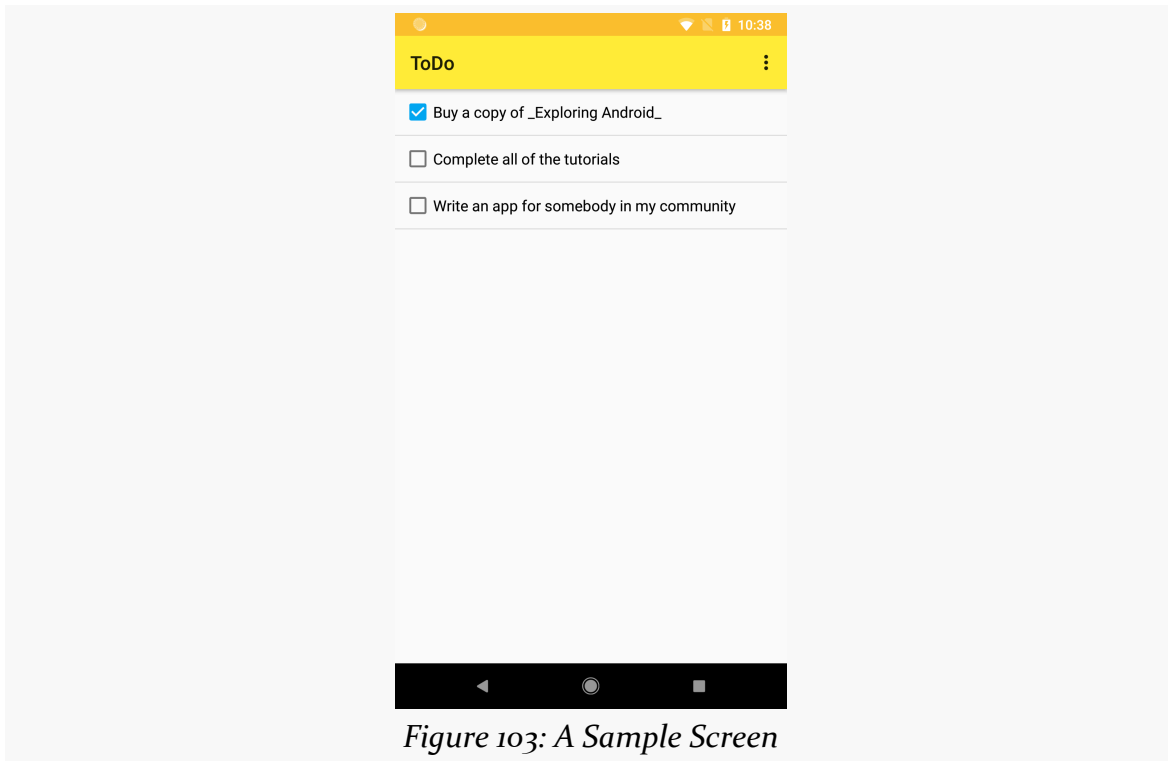
some basic scenarios for using ConstraintLayout, and look at another widget: EditText.

The Role of Containers

Some apps have screens that contain just one widget.

Not many, though.

Most of the time, our user interface is more complicated than that... even if it does not seem that complicated when you look at it:



Here we have:

- Three checkboxes
- Three labels adjacent to those checkboxes
- Some sort of title (“ToDo”)
- A strange little “...” icon in the upper right
- A yellow background behind the title and icon

This takes more than one XML element to represent in a layout resource. In fact, this takes more than one layout resource. This app is a “to-do list” sort of app, and it can handle an arbitrary number of to-do items. So while this screenshot shows three rows of checkboxes-and-labels, really the user could have 0 to N of them.

As we covered previously, containers extend from `ViewGroup`, and the job of a `ViewGroup` is to organize some number of children and display them on the screen. `ViewGroup` itself extends from `View`, so the children of a `ViewGroup` can be a mix of `View` and `ViewGroup` objects.

So, in this screenshot, we clearly have at least one `ViewGroup` that is organizing all of those widgets and displaying them.

Layouts and Adapter-Based Containers

Roughly speaking, there are two major types of `ViewGroup` implementations:

- There are those that organize a set of children known at the time that you are writing the app
- There are those that organize a set of children that cannot be known until the app is run

For example, if we were to write a to-do list app like the one shown above, we do not know how many to-do items there are in the list. That depends on how many the user enters that we save in a database, a Web service, or some other place. However, for each individual to-do list item, we know that the visual representation of that item is a checkbox and a label showing the task to be performed.

So, in this case, we have both types of `ViewGroup`. The list of to-do items is managed by a `ViewGroup` that is optimized for organizing data at runtime, while each row in that list is managed by a `ViewGroup` that is optimized for organizing data when we write the app.

The most popular `ViewGroup` for organizing data at runtime is `RecyclerView`, though there are other options. We will explore `RecyclerView` more [later in the book](#). The recommended `ViewGroup` for organizing widgets in layout resources is a `ConstraintLayout`, though there are other options.

ConstraintLayout: One Layout To Rule Them All

If you survey the Internet, you will find that ConstraintLayout is popular but relatively new. You will find references to other `...Layout` classes, such as `LinearLayout` and `RelativeLayout`. We will explore those briefly [later in this chapter](#). Suffice it to say that `LinearLayout` and `RelativeLayout` existed from the beginning (2008's Android 1.0), while ConstraintLayout debuted in 2016. And, there is nothing that ConstraintLayout can do that cannot also be accomplished by some mix of the classic layout classes (e.g., `LinearLayout`).

So, why did Google bother with 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.

However, in 2008, we had no drag-and-drop GUI builder. `LinearLayout`, `RelativeLayout`, and kin were not designed with drag-and-drop in mind. As it turns out, some of those (e.g., `LinearLayout`) work well in a drag-and-drop model, while others (e.g., `RelativeLayout`) do not.

By contrast, ConstraintLayout was designed with drag-and-drop GUI building in mind.

Performance

The classic layout classes were not written with performance in mind. Some of their features work well but are a bit slow to execute. They are not so slow as to be unusable, but they do make the app a bit more sluggish, particularly when scrolling the screen.

The objective of ConstraintLayout was to offer all of the capabilities of the classic containers — and more — while keeping performance in mind.

Library vs. Framework

LinearLayout, RelativeLayout, and the other classic containers are framework classes. As a result, they could have different implementations on different versions of Android. Since that causes a lot of pain for maintaining backwards compatibility, Google does not change these classes that often.

ConstraintLayout, by contrast, comes from a library. This has a cost, as now each app needs its own copy of the ConstraintLayout code. However, it also means that Google can keep improving ConstraintLayout, and developers can adopt newer versions of the library that add new features or fix bugs.

Getting ConstraintLayout

Since ConstraintLayout comes from a library, we need to ensure that we have this library in our module. If you create the project from the Android Studio new-project wizard, it is very likely that you already have the library in your dependencies.

What you are looking for is `androidx.constraintlayout:constraintlayout` for some version:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.1'  
    implementation "androidx.activity:activity:1.3.1"  
}
```

(from [ConstraintRow/build.gradle](#))

You may find that you have a different artifact, `com.android.support.constraint:constraint-layout`. This too provides ConstraintLayout, but it is the older artifact, from [the Android Support Library](#). If you have other artifacts in your dependencies that are androidx-based, you will want to use the androidx edition of the constraintlayout artifact.

Using Widgets and Containers from Libraries

In [an earlier chapter](#), we saw that layout resources use XML elements to describe the UI. The elements that we explored — TextView and Button — had simple element names, plus attributes that were prefixed with `android:`. This is how you will use

most of the widgets and containers that are part of the “framework classes” — the classes that are part of Android itself and ship with Android devices.

However, `ConstraintLayout` is from a library, so it is not part of the framework classes. A copy of the code for `ConstraintLayout` is packaged in your APK and ships with your app.

Using widgets and containers from libraries is very similar to using widgets and containers from the framework classes, with two key differences.

Fully-Qualified Class Name

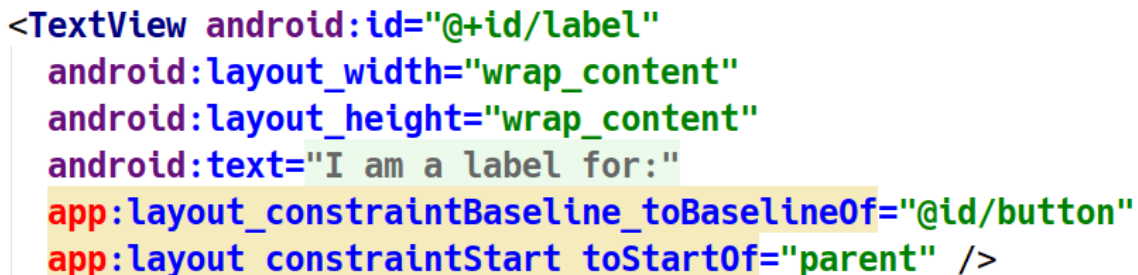
In [the chapter on widgets](#), we focused on `TextView` and `Button`. Those widgets have XML elements named `<TextView>` and `<Button>`, respectively.

Most framework widgets work that way: the XML element can be the bare class name. No library widgets work that way. For those, the XML element needs to be the *fully-qualified* class name. So, when using `ConstraintLayout`, you will see XML elements like `<androidx.constraintlayout.widget.ConstraintLayout>`. This gets a bit wordy if you are working with the XML, but if you are mostly using the drag-and-drop GUI builder, you will barely notice the difference.

app: Attributes

You will also find that some of the XML attributes do not get the `android:` prefix, but instead get an `app:` prefix. These are XML attributes defined by the library code, whereas `android:-`prefixed attributes come from framework code.

This means that the first time you start using library widgets and containers in a layout resource, you might see `app` show up in red:



```
<TextView android:id="@+id/label"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="I am a label for:"  
    app:layout_constraintBaseline_toBaselineOf="@id/button"  
    app:layout_constraintStart_toStartOf="parent" />
```

Figure 104: Android Studio Layout XML Editor, Showing Red app

This means that the app XML namespace has not been declared. The simplest way to add it is to:

- Put your text cursor somewhere in the red-highlighted app text
- Press Alt-Enter (or the equivalent for macOS) to bring up the “quick-fix” menu:

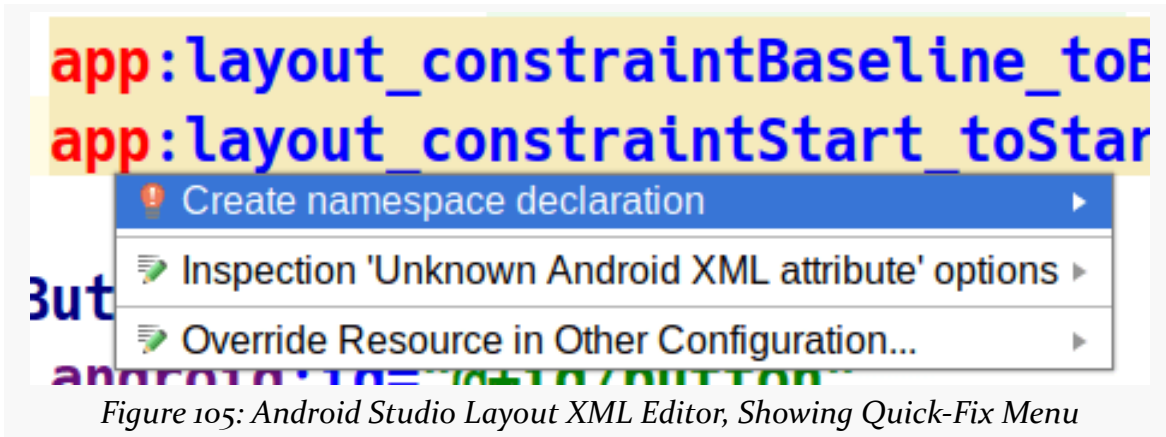


Figure 105: Android Studio Layout XML Editor, Showing Quick-Fix Menu

- Choose “Create namespace declaration” from the menu

This will add `xmlns:app="http://schemas.android.com/apk/res-auto"` to your root XML element of the layout resource.

A Quick RTL Refresher

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. LTR, in turn, refers to left-to-right languages.

Android supports both LTR and RTL. We noted in [the chapter introducing the manifest](#) how your app can advertise that it supports RTL. This, in turn, will cause you to refer to “start” and “end” instead of “left” and “right” when positioning widgets on the screen:

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.

Simple Rows with ConstraintLayout

The ConstraintRow sample module in the [Sampler](#) and [SamplerJ](#) projects is very similar to the SimpleText sample. The layout now has a ConstraintLayout wrapped around a TextView and a Button.

The XML

Our root element is now

`<androidx.constraintlayout.widget.ConstraintLayout>`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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:padding="@dimen/container_padding">

    <TextView android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/label_caption"
        app:layout_constraintBaseline_toBaselineOf="@id/button"
        app:layout_constraintStart_toStartOf="parent" />

    <Button
        android:id="@+id/button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="@string/button_caption"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toEndOf="@id/label"
```

INTRODUCING CONSTRAINTLAYOUT

```
app:layout_constraintEnd_toEndOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ConstraintRow/src/main/res/layout/activity_main.xml](#))

We give that `ConstraintLayout` a height and width of `match_parent`, which indicates that this widget or container should fill up the available space of its parent container. In this case, the “parent” is a container that occupies most of the space on the screen, so the `ConstraintLayout` will fill that space. We also give the `ConstraintLayout` padding on all four sides, so the contents of the `ConstraintLayout` will be inset from the edges by however much `@dimen/container_padding` calls for (8dp in this case).

There are two XML elements that are contained in the `ConstraintLayout`: our `TextView` and our `Button`. Both have IDs, sizes, and captions (via `android:text`). However, both also have anchoring rules, indicating where the widgets should be positioned within the `ConstraintLayout`.

The `TextView` has `app:layout_constraintStart_toStartOf="parent"`. All attributes starting with `app:layout_constraint` are rules for children of `ConstraintLayout`. `app:layout_constraintStart_toStartOf="parent"` says:

- We are trying to constrain the “start” edge of the widget (`constraintStart`), where the “start” edge is on the left for left-to-right languages and on the right for right-to-left languages
- We are trying to anchor that edge to the start edge of something else (`toStartOf`)
- The “something else” is the `ConstraintLayout` itself (“parent”)

So, `app:layout_constraintStart_toStartOf="parent"` will anchor the start edge of the `TextView` to the start side of the `ConstraintLayout`.

Similarly, the `Button` has:

- `app:layout_constraintTop_toTopOf="parent"` to anchor the top of the `Button` to the top of the `ConstraintLayout`
- `app:layout_constraintEnd_toEndOf` to anchor the “end” of the `Button` to the “end” of the `ConstraintLayout`, where “end” is on the right for left-to-right languages and on the left for right-to-left languages

The `TextView` also has `app:layout_constraintBaseline_toBaselineOf="@id/`

button". Here, "baseline" refers to the invisible line that text appears to "sit" upon. A `TextView` — and any subclasses, like `Button` — has a baseline. Here, we are anchoring the baseline of the `TextView` to the baseline of the `Button`, so wherever the `Button` winds up, the `TextView` will have a matching vertical position. The `"@id/button"` ties into the `@+id/button` declared in the `android:id` attribute of the `Button`, so the `ConstraintLayout` knows what widget is the target of this anchoring rule.

The `Button` also has `app:layout_constraintStart_toEndOf="@id/label"`. This says that we want to anchor the start edge of the `Button` to the end edge of the `TextView` (whose `android:id` value is `@+id/label`).

The `Button` has an unusual `android:layout_width` value: `0dp`. Normally, this would mean a width of zero `dp`, which would be a bit short. For a child of `ConstraintLayout`, though, `0dp` means "the width is determined by the horizontal constraints". In the case of the `Button`:

- The start edge is anchored to the end edge of the `TextView`
- The end edge is anchored to the end edge of the `ConstraintLayout`

As a result, with a width of `0dp`, the `Button` will be *stretched* to fill the space between those two anchor positions. If instead the width were `wrap_content`, the `Button` width would be determined by its caption, and it would be *centered* in between those two anchor positions.

The Android Studio Graphical Layout Editor

If you click on a widget in the `ConstraintLayout` in the blueprint view, that view will show squares on the corners and circles centered on the edges:



Figure 106: Android Studio Graphical Layout Editor, Blueprint View

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.

The circles are more important, as they allow you to define the constraints, by dragging a circle to some anchor point:



Figure 107: Blueprint View, Showing Constraint Being Created

You can drag the circle to an equivalent circle on another widget or to the edges of the ConstraintLayout, to establish an anchoring rule between those two points. That rule is represented by an arrow connecting the two widgets, with the start and end of the arrow showing the sides that were constrained.

The Result

If you run either edition of the ConstraintRow sample, you will see the result matches the graphical layout editor and what we asked for in the XML attributes:

- The TextView is aligned on the start edge of the screen and is vertically aligned with the baseline of the Button
- The TextView takes up its “natural” space based upon its caption
- The Button fills the space between the TextView and the end edge of the screen
- The Button is aligned with the top of the screen
- Everything is inset 8dp from the edges

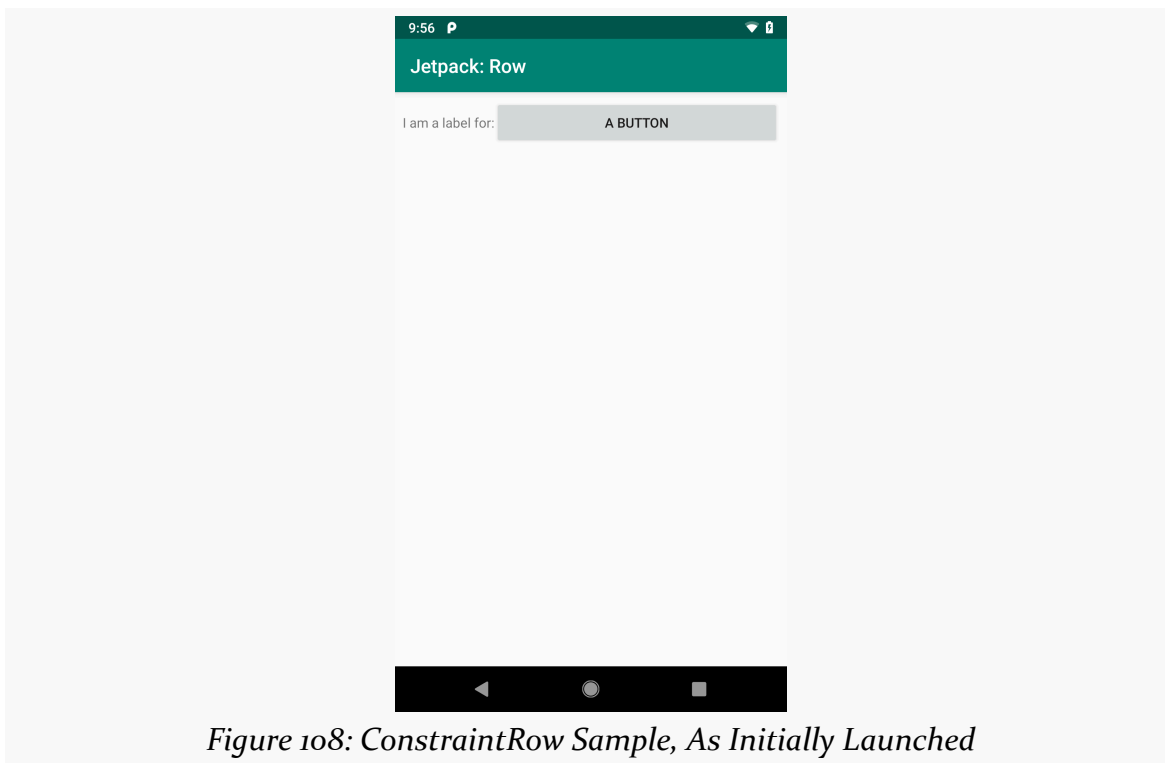


Figure 108: ConstraintRow Sample, As Initially Launched

Starting from Scratch

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 `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:

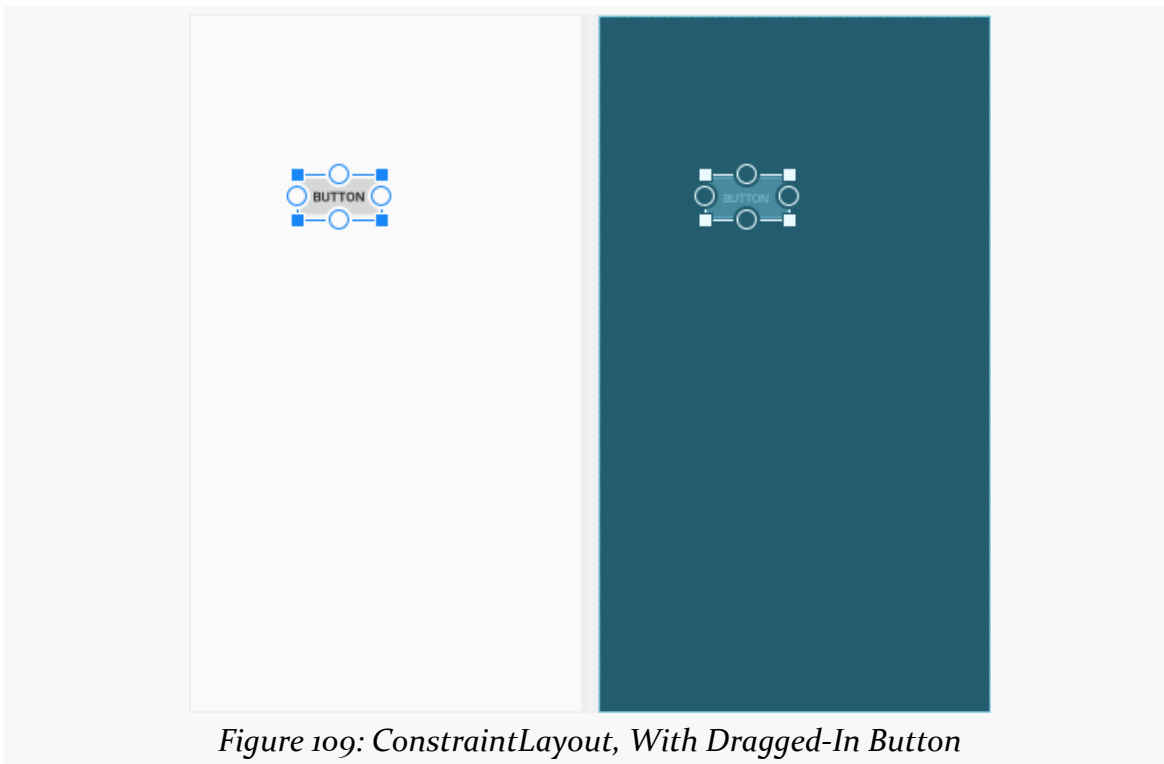


Figure 109: ConstraintLayout, With Dragged-In Button

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"?>
<androidx.constraintlayout.widget.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">

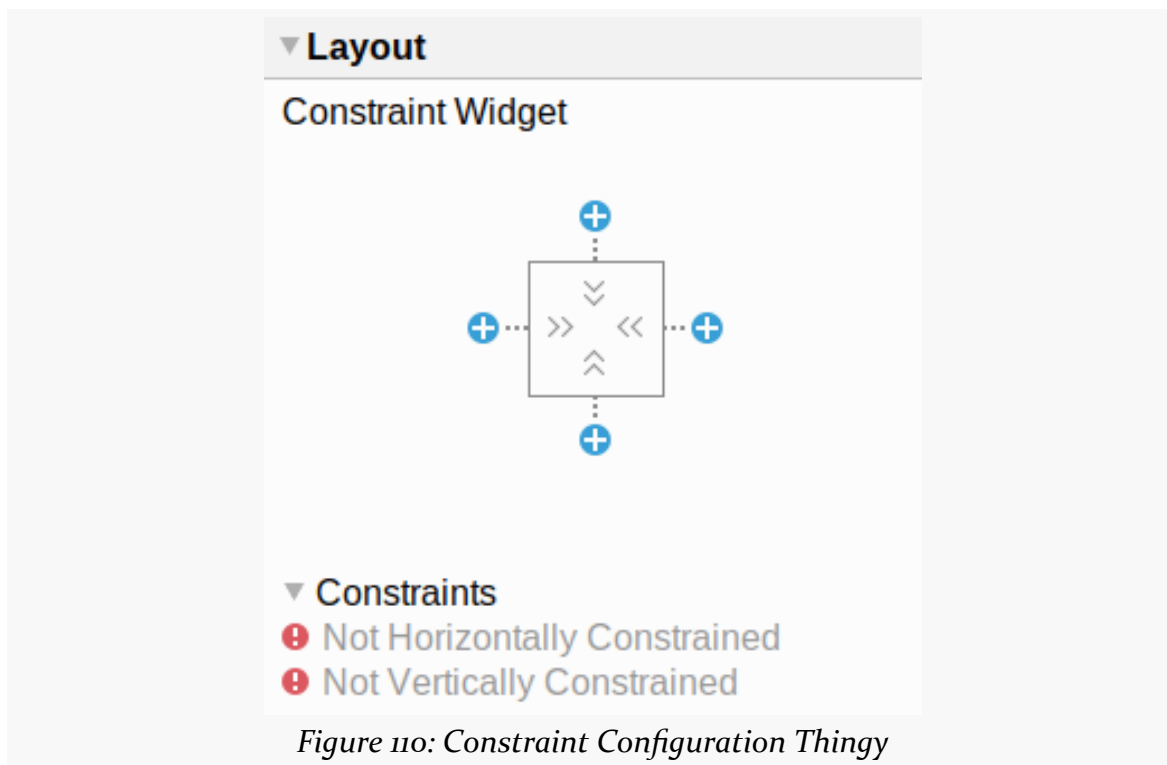
    <Button
        android:id="@+id/button9"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        tools:layout_editor_absoluteX="77dp"
        tools:layout_editor_absoluteY="36dp" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

As we discussed earlier in the book, 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, dragging the circles on the widget's edges and connecting them to the edges of other widgets or to the ConstraintLayout itself.

ConstraintLayout and the Attributes Pane

When you click on a widget inside a ConstraintLayout, the Attributes pane has a strange-looking control in the “Layout” category of attributes:



The chevrons inside the square indicate that the sizing rule is `wrap_content` for each axis. Clicking on one of the chevrons will toggle between states for that axis:

- a fixed width, indicated by a sizing bar
- 0dp, for stretching the size to the available space, indicated by a sawtooth line
- wrap_content, indicated by those chevrons

EditText: Making Users Type Stuff

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?).

Graphical Layout Editor

The Palette in the graphical layout editor has a whole section dedicated primarily to `EditText` widgets, named “Text”:

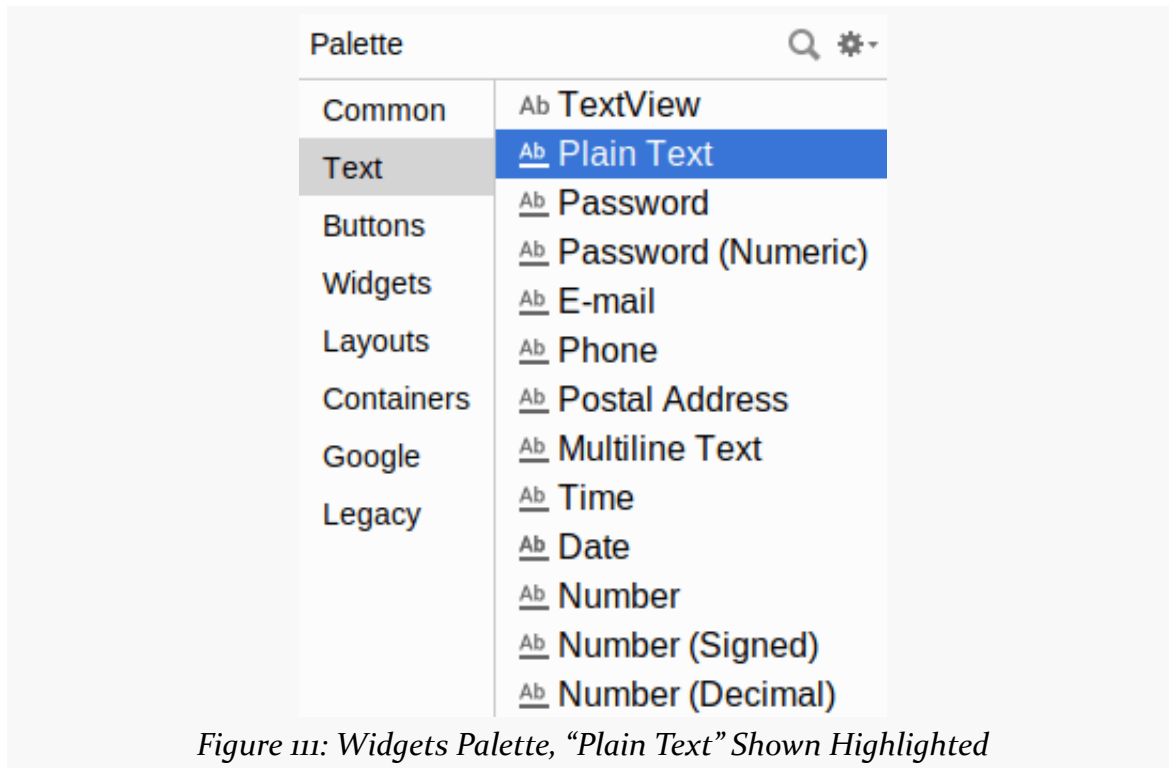


Figure 111: Widgets Palette, “Plain Text” Shown Highlighted

The first entry is a `TextView`. The second entry (“Plain Text”) 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, or edit the `EditText` XML as you see fit. The “Id” and “Text” attributes are the same as found on `TextView`, as are many other attributes, inherited from `TextView`.

Notable 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:

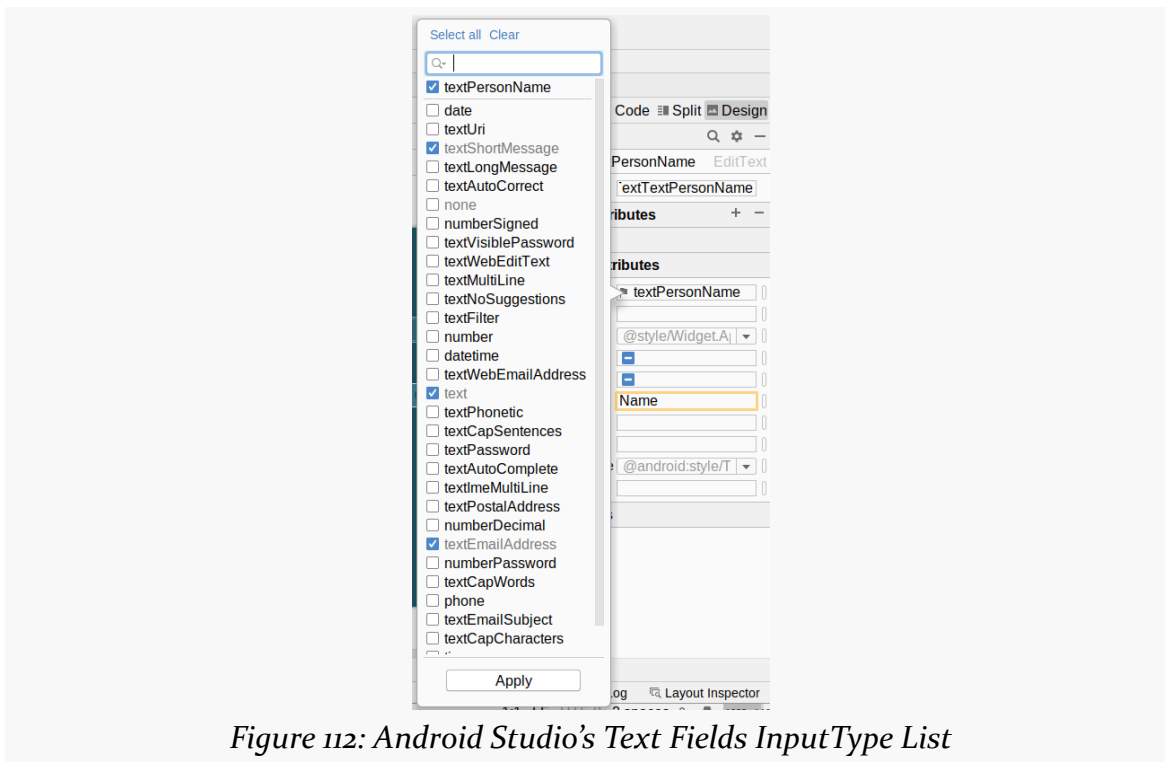


Figure 112: Android Studio’s Text Fields InputType List

More Complex Forms

Let’s look at a bit more elaborate example, found in the `ConstraintForm` modules of the [Java](#) and [Kotlin](#) projects.

What We Want

We want to have a form that collects the first and last name of a person, with a Button to submit the form. We are not reacting to that button click — the purpose of this sample is to see how the form is constructed.

This seems straightforward enough. For example, we can have a `ConstraintLayout` that collects the first name and last name in two rows, with the Button below the

INTRODUCING CONSTRAINTLAYOUT

last name row.

However, if we limit ourselves to the sort of `ConstraintLayout` anchoring rules that we have seen above, we would wind up with a layout that looks like this:

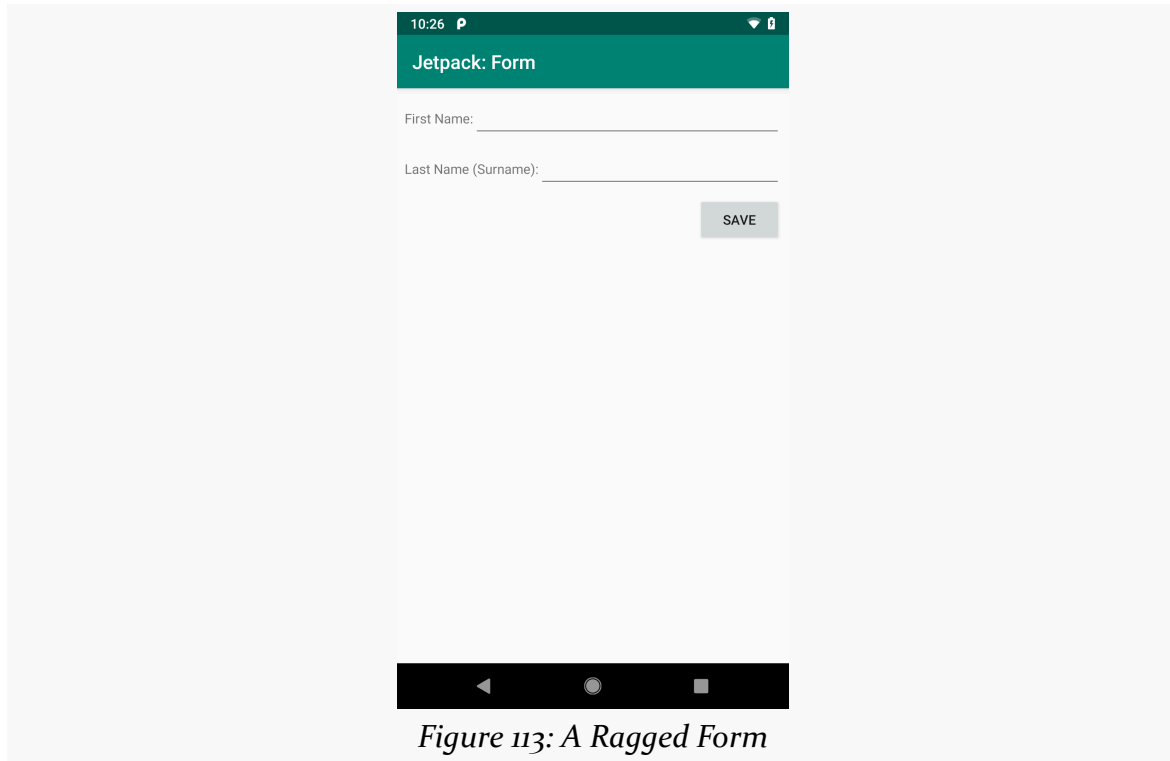


Figure 113: A Ragged Form

INTRODUCING CONSTRAINTLAYOUT

This works, but it would be nicer if we had columns for the labels and the fields:

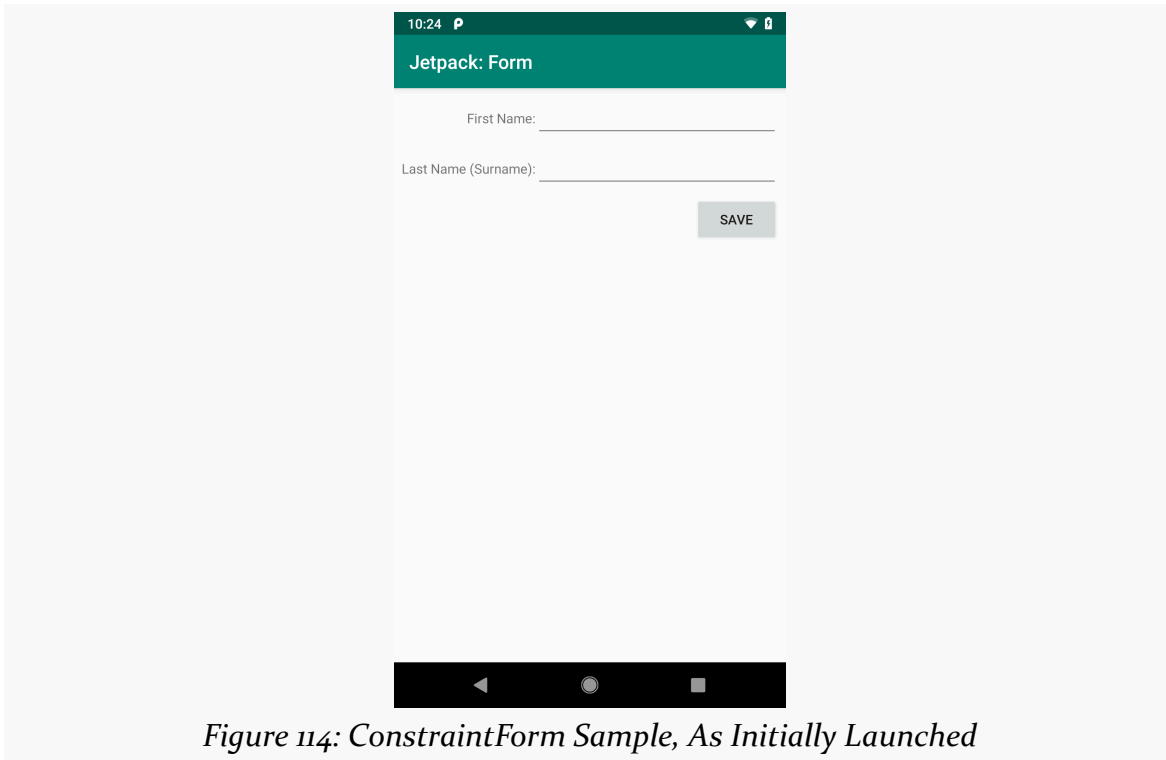


Figure 114: ConstraintForm Sample, As Initially Launched

Fortunately, `ConstraintLayout` offers some features for helping us get this sort of look.

How We Get There

Our layout is more complex than the earlier example, owing in part to having five children of the `ConstraintLayout`: two `TextView` widgets, two `EditText` widgets, and a `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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:padding="@dimen/container_padding">

    <androidx.constraintlayout.widget.Barrier
        android:id="@+id/barrier"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/margin_barrier"
    android:layout_marginEnd="@dimen/margin_barrier"
    app:barrierDirection="end"
    app:constraint_referenced_ids="labelFirst,labelLast" />

<TextView
    android:id="@+id/labelFirst"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:labelFor="@id/firstName"
    android:text="@string/label_first"
    app:layout_constraintEnd_toStartOf="@id/barrier"
    app:layout_constraintBaseline_toBaselineOf="@id/firstName"
    app:layout_constraintHorizontal_bias="1.0"
    app:layout_constraintStart_toStartOf="parent" />

<EditText
    android:id="@+id/firstName"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:inputType="textPersonName"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@id/barrier"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/labelLast"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:labelFor="@id/lastName"
    android:text="@string/label_last"
    app:layout_constraintEnd_toStartOf="@id/barrier"
    app:layout_constraintBaseline_toBaselineOf="@id/lastName"
    app:layout_constraintHorizontal_bias="1.0"
    app:layout_constraintStart_toStartOf="parent" />

<EditText
    android:id="@+id/lastName"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:inputType="textPersonName"
    android:layout_marginTop="@dimen/margin_row"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@id/barrier"
    app:layout_constraintTop_toBottomOf="@id/firstName" />
```

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/save_caption"
    android:layout_marginTop="@dimen/margin_row"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/lastName" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ConstraintForm/src/main/res/layout/activity_main.xml](#))

If you look at that XML, there are six child elements of the `ConstraintLayout` — we will explore that sixth one shortly.

Naming the Widgets

Each of the widgets gets a fairly simple `android:id`:

- `firstName` and `lastName` for the two `EditText` widgets
- `firstLabel` and `lastLabel` for the two `TextView` widgets
- `button` for the `Button`

Since we are not doing anything with these widgets from our Java/Kotlin code, the widget IDs are purely for internal use within this layout resource.

Barrier: You Shall Not Pass

The sixth child of the `ConstraintLayout` is a `Barrier`. `Barrier` is one of a couple of options with `ConstraintLayout` for creating a virtual anchor point: a place for you to reference in layout anchoring rules that is not itself a widget in the layout.

Specifically, this `Barrier` has:

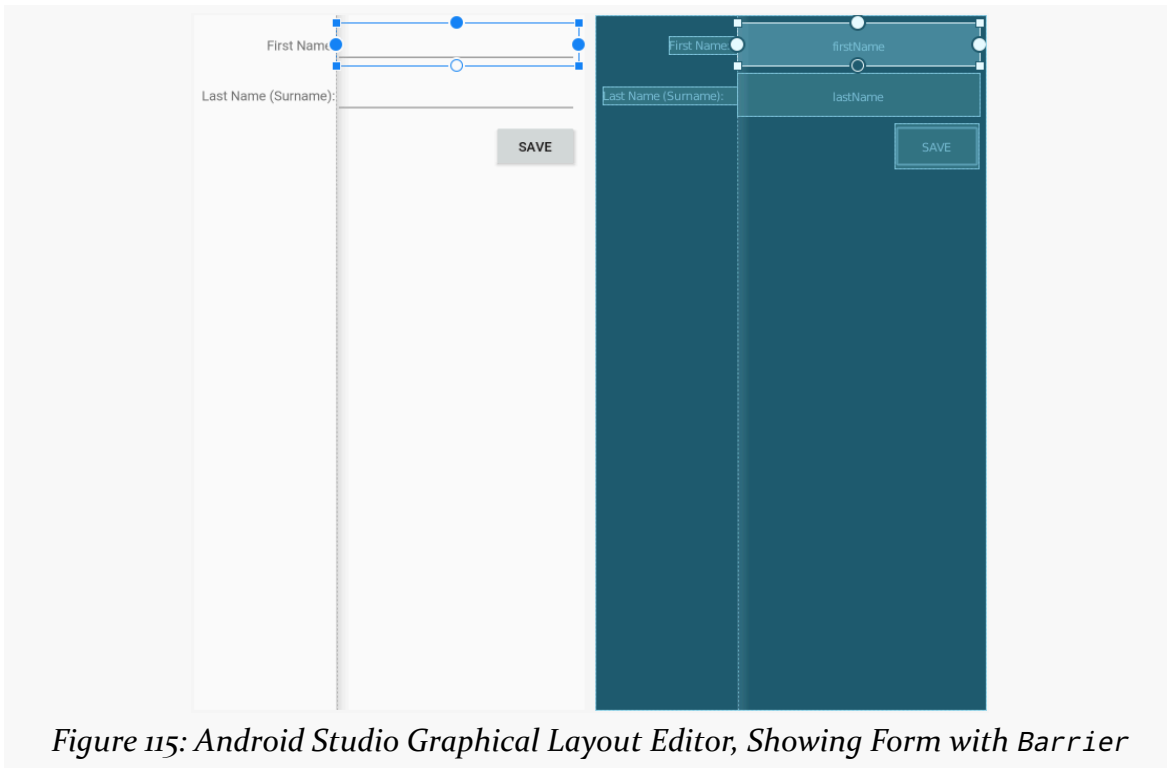
- `app:barrierDirection="end"`
- `app:constraint_referenced_ids="labelFirst,labelLast"`

Here, the “barrier direction” indicates where the barrier is established relative to the widgets referenced in the `app:constraint_referenced_ids` attribute. In our case, the end for `app:barrierDirection` means “put this `Barrier` at the end position of the child that is farthest from the start position”. Both of the `TextView` widgets are set up to start from the start edge of the `ConstraintLayout` and take up as much

INTRODUCING CONSTRAINTLAYOUT

room as their content requires (`android:layout_width="wrap_content"`). As a result, the Barrier is placed at the end of whichever of those two `TextView` widgets is longest. In the sample app, that is the second label, which uses `Last Name (Surname)` as the caption (courtesy of `@string/label_last`). So, the Barrier is placed where the `labelLast` `TextView` ends.

We then set up the `firstName` and `lastName` widgets to have their start position be where the Barrier is, via `app:layout_constraintStart_toEndOf="@id/barrier"`. This is why both fields have the same starting position, and that position is determined by the longest of the labels.



As a result, our four main widgets — the two labels and the two fields — are organized into two columns, where the width of the first column is determined by the width of the labels inside of it.

Your Position Shows Some Bias

Our two labels do not have the same width. The `labelLast` text was set up to be excessively long to illustrate this.

INTRODUCING CONSTRAINTLAYOUT

By default, if we just anchored the labels to the start edge of the `ConstraintLayout`, their text would wind up on that edge. That is a bit awkward, though, in that our `labelFirst` text would wind up relatively far away from its associated field.

To address this, our layout has each label do two things:

1. Constrain both the start and end edges, where the end is anchored to the `Barrier` via `app:layout_constraintEnd_toStartOf="@id/barrier"`
2. Shove the text to the end side via `app:layout_constraintHorizontal_bias="1.0"`

Bias, in terms of `ConstraintLayout`, means “if there is extra room, slide the widget in this direction along the axis”. The default bias is 0.5, meaning that the widget is centered in the available space. For the horizontal axis, 0.0 means “slide the widget all the way towards the start side” and 1.0 means “slide the widget all the way towards the end side”.

By having the label end at the `Barrier` and having its associated field start at the `Barrier`, we ensure that the label and the field are close together.

Declaring the Rows

In terms of the rows, the fields drive the vertical positioning:

- The top of `firstName` is anchored to the top of the `ConstraintLayout` via `app:layout_constraintTop_toTopOf="parent"`
- The top of `lastName` is anchored to the bottom of `firstName` via `app:layout_constraintTop_toBottomOf="@id/firstName"`
- The top of `button` is anchored to the bottom of `lastName` via `app:layout_constraintTop_toBottomOf="@id/lastName"`
- The labels have their baselines aligned with their associated fields

In addition:

- Each label has `android:labelFor` set, pointing to its associated field, for the benefit of screen readers and other assistive technologies
- The `Button` has `app:layout_constraintEnd_toEndOf="parent"` and no start-side anchor, so it will be flush on the end side of the `ConstraintLayout`

Turning Back to RTL

In order for the “start”/“end” attributes to reverse their positions based on language direction, 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:

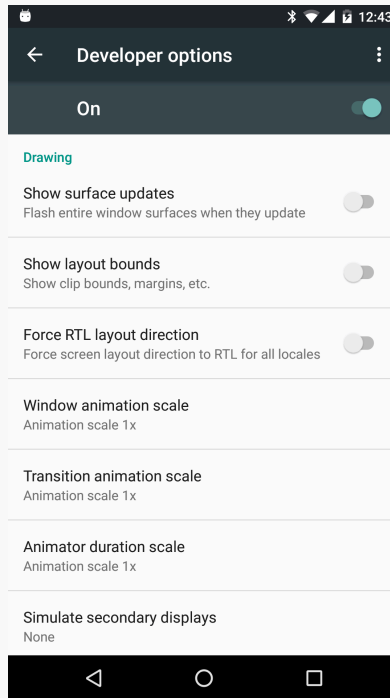


Figure 116: Developer Options in Settings, Normal Mode

INTRODUCING CONSTRAINTLAYOUT

Tapping that switch uses RTL layout rules — with “start” referring to the right and “end” referring to the left — for all languages:

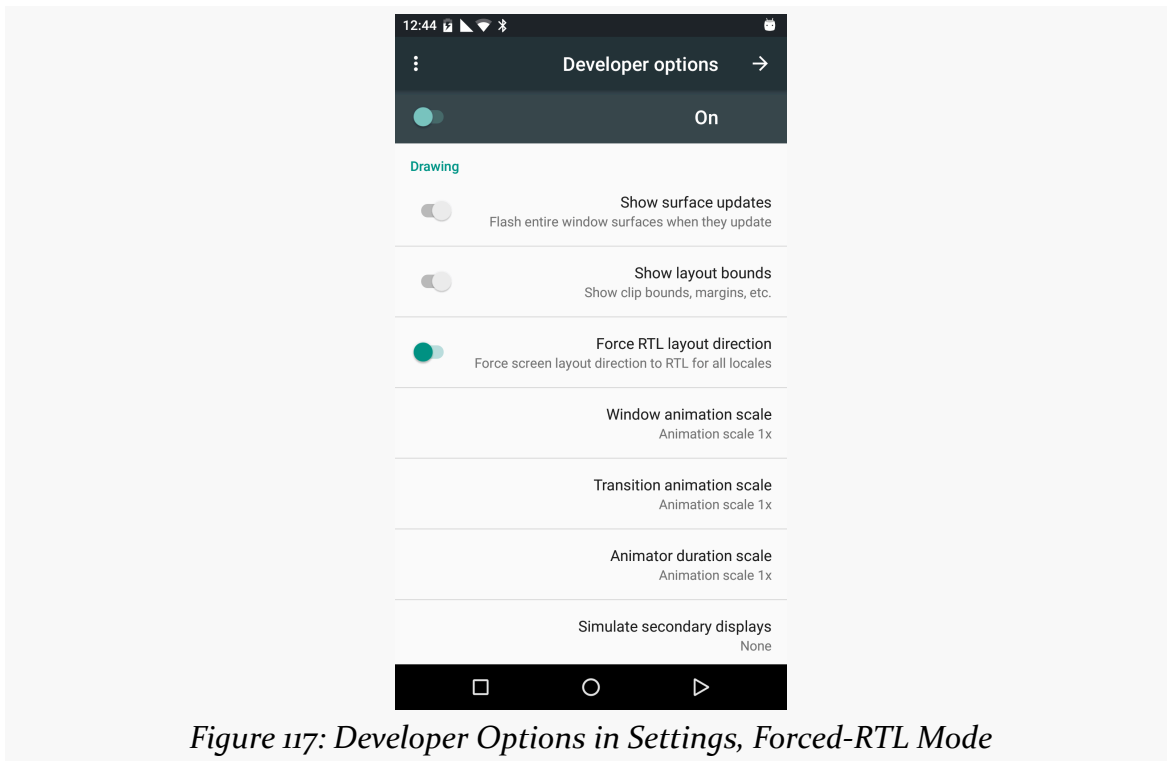


Figure 117: Developer Options in Settings, Forced-RTL Mode

More Fun with ConstraintLayout

There are many more capabilities of ConstraintLayout than what we have covered here. You can:

- Use Guideline to set up anchoring points, a bit like Barrier, but set at arbitrary locations
- Use `app:layout_constraintWidth_percent` and `app:layout_constraintHeight_percent` to size children as a percentage of the ConstraintLayout size along a particular axis
- Use `app:layout_constraintDimensionRatio` to force a child to adhere to a particular aspect ratio
- Use ConstraintSet to alter the anchoring and sizing rules for children of ConstraintLayout at runtime
- And more!

We will see `ConstraintLayout` used in many of the samples through the rest of the book.

Notes on the Classic Containers

While `ConstraintLayout` is the recommended solution for Jetpack-based Android app development, it is not the only option. And since `ConstraintLayout` is relatively new, there is a lot of existing code that does not use it.

Four container classes were the dominant options starting with Android 1.0 and can still be used today: `LinearLayout`, `RelativeLayout`, `TableLayout`, and `FrameLayout`.

LinearLayout

`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. Since many other GUI toolkits use this sort of an approach, a lot of developers used `LinearLayout` extensively. Plus, it is *very* easy to use: other than an `android:orientation` attribute to indicate if it should be a row (`horizontal`) or column (`vertical`), nothing else is required.

And, for fairly simple scenarios, there is nothing wrong with using `LinearLayout`.

However, `LinearLayout` was designed around simple usage and simple implementation, not performance. Having lots of nested `LinearLayout` containers can slow things down. And for a complex form, you would need lots of `LinearLayout` containers. Even something as simple as `ConstraintForm` would take three `LinearLayout` instances:

- Two horizontal ones for the rows containing the labels and fields
- A vertical one for holding those rows plus the Button

RelativeLayout

On the surface, `RelativeLayout` looks a lot like `ConstraintLayout`. `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. And it does so via special attributes on the children, saying what they are anchored to.

All of that sounds like `ConstraintLayout`.

However, `RelativeLayout` was not created with performance or drag-and-drop GUI builders in mind. `ConstraintLayout` has a superset of `RelativeLayout` features, and `RelativeLayout` has its own set of attributes similar to, but distinct from, those in `ConstraintLayout`. There is little reason to use `RelativeLayout` today.

TableLayout

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. Using a `TableLayout` with `TableRow` works a lot like using an HTML `<table>` with `<tr>` elements.

When Android started out, HTML tables were very popular, not only for tabular data but for general page layout. CSS had only caught on in recent years, so lots of Web developers were used to using HTML tables to try to position things on the screen in the desired locations. Having a `TableLayout` that mimicked HTML tables was a logical move, to help ease the transition for those early Web designers.

With `Barrier`, you can set up a `ConstraintLayout` that can handle table structures, with rows and columns, where the columns can vary in width based on contents. However, it is likely that there will be some scenarios that would be very difficult to implement with `ConstraintLayout` that `TableLayout` could handle easily. Overall, though, `ConstraintLayout` is much more powerful than is `TableLayout`. Plus, `TableLayout` suffers from the same performance problems of `LinearLayout`, with lots of nested containers.

So, in general, if you see a grid sort of structure, try using a `ConstraintLayout`, and consider falling back to `TableLayout` only if needed.

FrameLayout

Android has a `FrameLayout` class. Like `ConstraintLayout`, `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. Children can overlap on the Z axis (which `ConstraintLayout` and `RelativeLayout` also support)

The result is that all the widgets are stacked one on top of another.

This may seem useless.

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/Kotlin 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.

Integrating Common Form Widgets

TextView, Button, and EditText form the foundation of many user interfaces in Android, and their analogues form the foundation of other GUI toolkits.

Overall, the Android SDK has a reasonable range of widgets to choose from, though not everything is covered. In this chapter, we will review a number of other commonly-used widgets in the Android SDK and explore some other capabilities of Android's UI system.

All of the code in this chapter comes from the FormWidgets module of the [Java](#) and [Kotlin](#) sample projects.

ImageView and ImageButton

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 a layout resource) to specify what picture to use. These usually reference a drawable resource (e.g., `@drawable/icon`) or sometimes a mipmap resource (e.g., `@mipmap/ic_launcher`).

ImageButton, a subclass of ImageView, mixes in the standard Button behaviors. While both ImageView and ImageButton can call your code when they are clicked, ImageButton has built-in logic to visually respond to the click, the way that Button does.

Our sample app has an ImageView named `icon` and an ImageButton named `button`:

INTEGRATING COMMON FORM WIDGETS

```
<ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:contentDescription="@string/icon_caption"
    android:src="@mipmap/ic_launcher"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/log" />

<ImageButton
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:contentDescription="@string/button_caption"
    android:src="@mipmap/ic_launcher"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/icon" />
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

INTEGRATING COMMON FORM WIDGETS

These then show up in our overall activity's output, which stacks a bunch of widgets in a `ConstraintLayout`:

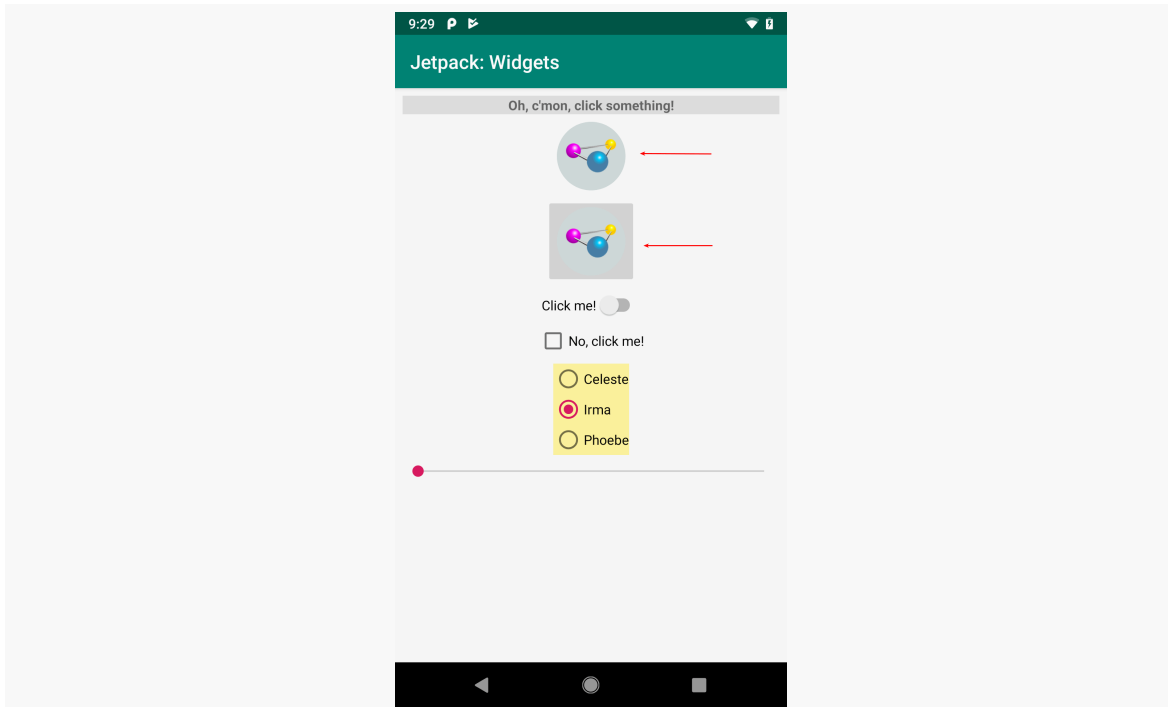


Figure 118: FormWidgets Sample, with ImageView and ImageButton Highlighted

Android Studio Graphical Layout Editor

The `ImageView` widget can be found in the “Common” portion of the Palette in the Android Studio graphical layout editor:

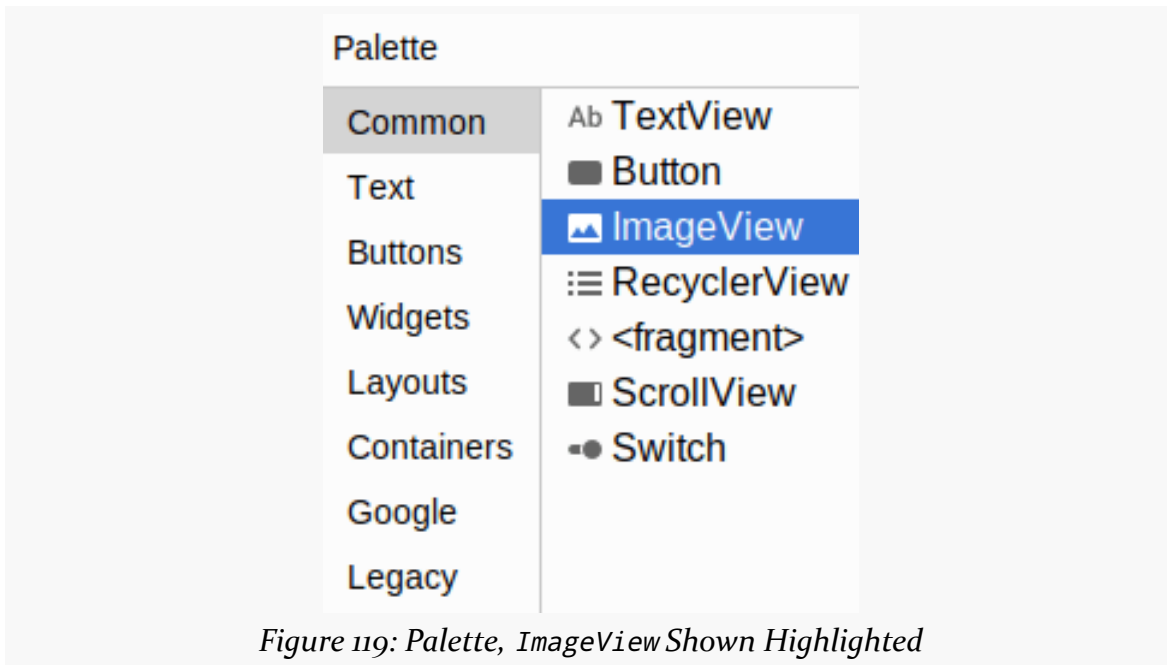
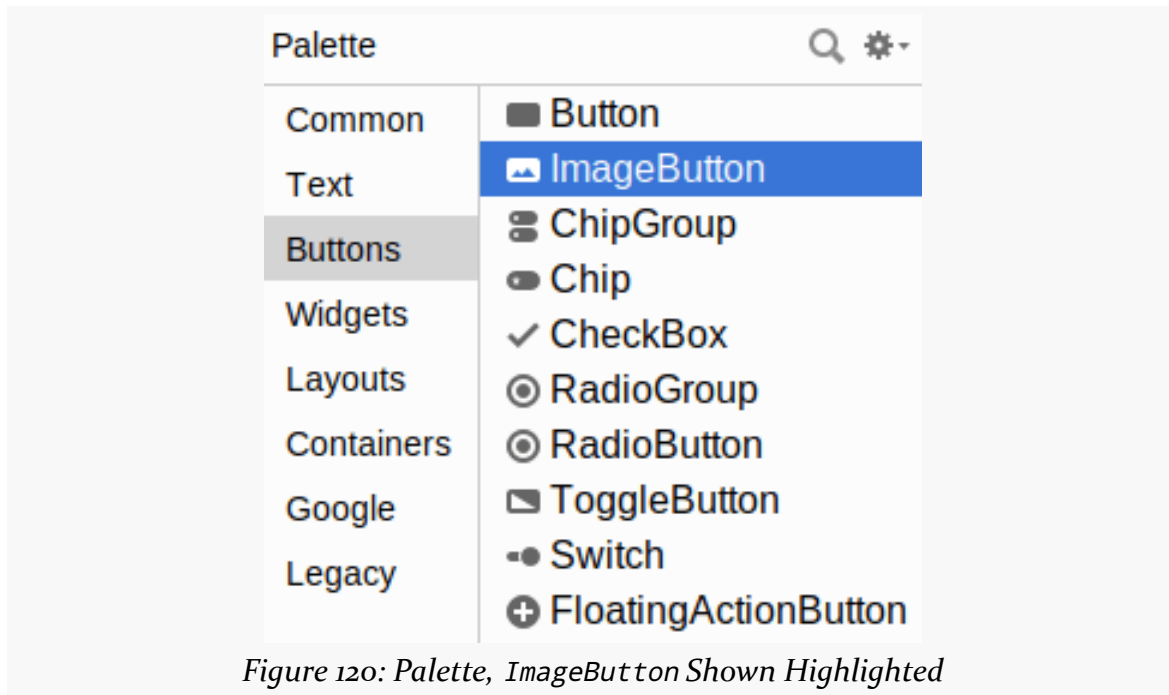


Figure 119: Palette, ImageView Shown Highlighted

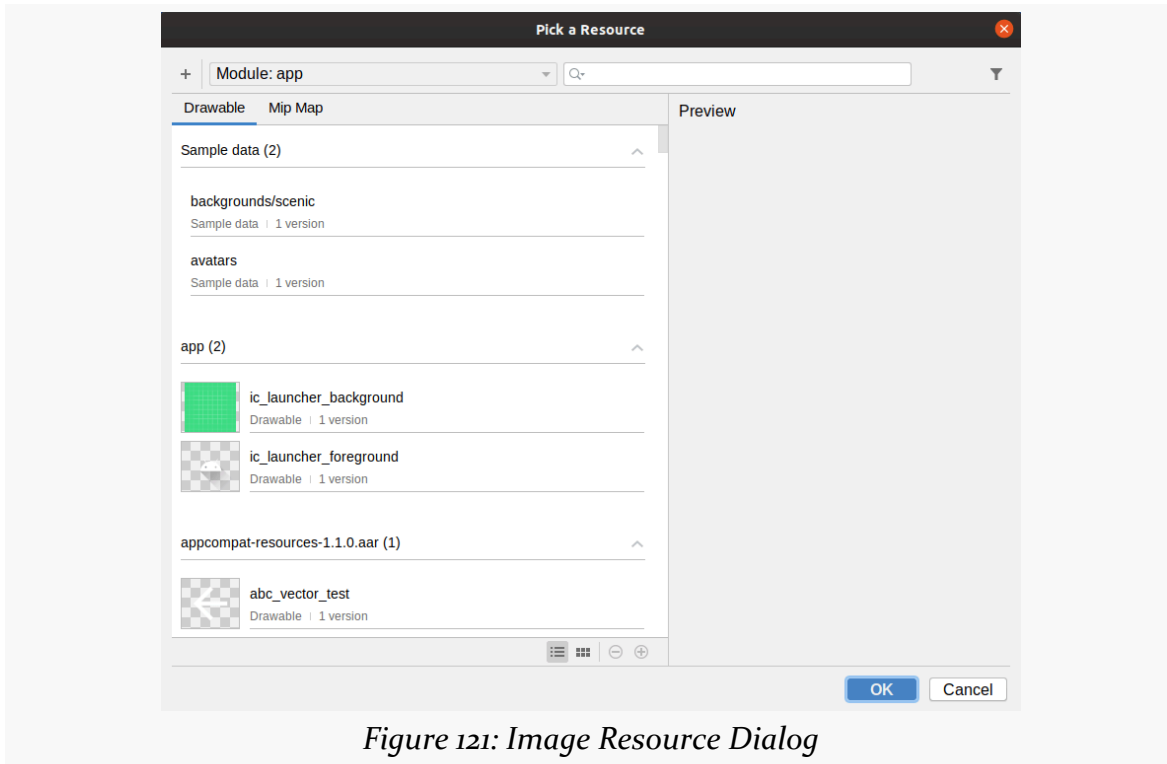
INTEGRATING COMMON FORM WIDGETS

ImageButton shows up in the “Buttons” portion:



INTEGRATING COMMON FORM WIDGETS

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:



Unfortunately, you have no choice but to choose one of these, as due to [some curious design decisions by Google](#), 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`:

- `srcCompat` 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 (note: `src` also works)
- `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

If you choose an image from the “Sample data” section in the resource dialog,

instead of `app:srcCompat`, you get `tools:srcCompat`. This provides an image that you can use in the IDE, but that image will not be displayed at runtime. This is useful for cases where you want to supply the image from a URL or something else dynamic.

Reacting to Events

You can call `setOnClickListener()` on an `ImageView` or `ImageButton` to find out when the user clicks the widget and do something:

```
binding.icon.setOnClickListener(v -> log(R.string.icon_clicked));
binding.button.setOnClickListener(v -> log(R.string.button_clicked));
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

```
binding.icon.setOnClickListener { log(R.string.icon_clicked) }
binding.button.setOnClickListener { log(R.string.button_clicked) }
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](#))

We will cover the `log()` function that both of those are using [a bit later in this chapter](#).

Compound Buttons

The Android SDK has a `CompoundButton` class that represents a widget that can be clicked to toggle a “checked” state. `CompoundButton` itself is an abstract class, but there are a few subclasses of `CompoundButton` of interest to Android app developers.

Switch

For a simple widget to show a checked or unchecked state, the modern approach is to use a `Switch`.

`Switch` extends `CompoundButton`, which in turn inherits from `TextView`. As a result, you can use `TextView` properties like `android:textColor` to format the widget, in addition to `android:text` to set the caption that should appear adjacent to the actual “switch” UI.

Within your Java/Kotlin code, you can call:

1. `isChecked()` to determine if the checkbox has been checked

INTEGRATING COMMON FORM WIDGETS

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, inverting whatever its current state is

Also, you can register a listener object (in this case, an instance of `OnCheckedChangeListener`) to be notified when the state of the checkbox changes.

Our sample app has an `Switch` named `swytch`:

```
<Switch
    android:id="@+id/swytch"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:text="@string/switch_caption"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/button" />
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

This appears beneath the `ImageButton`.

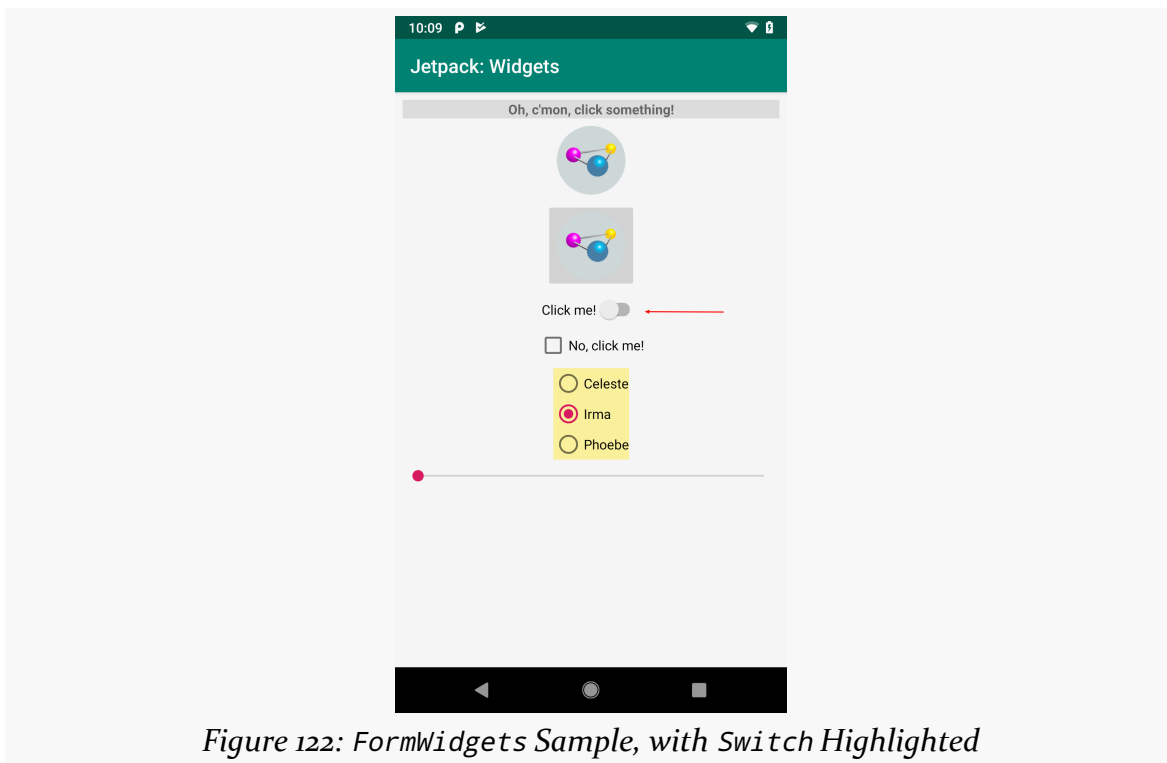
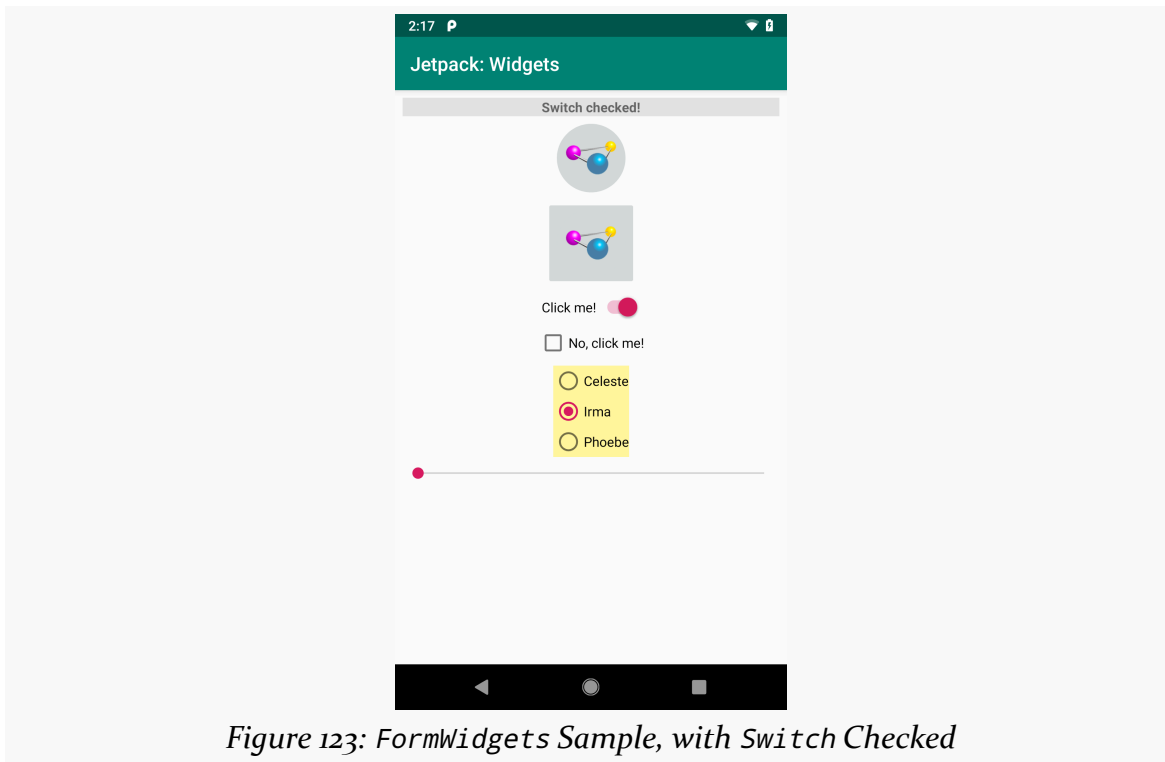


Figure 122: FormWidgets Sample, with Switch Highlighted

INTEGRATING COMMON FORM WIDGETS

In our case, initially it is unchecked, but the user can tap on it to check it:



Android Studio Graphical Layout Editor

The Switch widget can be found in the “Buttons” portion of the Palette in the Android Studio Graphical Layout editor:

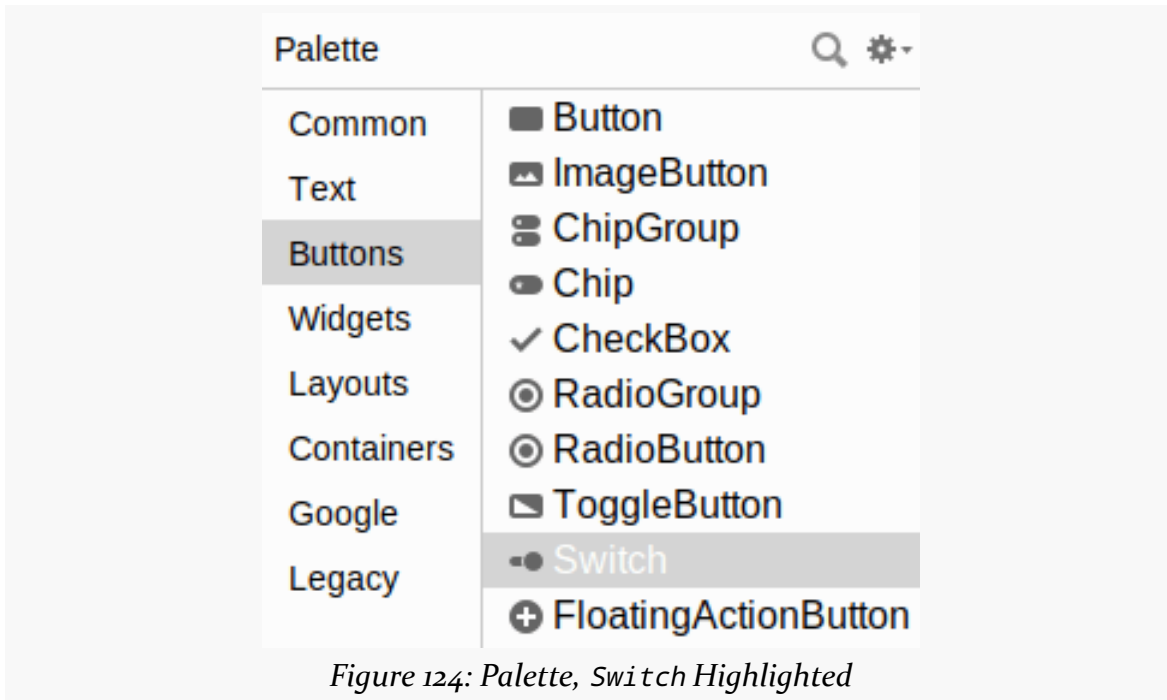


Figure 124: Palette, Switch Highlighted

You can drag one into the layout and configure it as desired using the Attributes pane. Mostly the attributes match those of `TextView` and `Button`.

Reacting to Events

The primary event listener for any form of `CompoundButton` is for changes in the checked state. You can register a `CompoundButton.OnCheckedChangeListener` via `setOnCheckedChangeListener()` from Java:

```
binding.swytch.setOnCheckedChangeListener((v, isChecked) ->
    log(isChecked ? R.string.switch_checked : R.string.switch_unchecked));
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

...or Kotlin:

INTEGRATING COMMON FORM WIDGETS

```
binding.swytch.setOnCheckedChangeListener { _, isChecked ->
    log(if (isChecked) R.string.switch_checked else R.string.switch_unchecked)
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](#))

In this case, our lambda expressions are converted into instances of `OnCheckedChangeListener`, with the lambda expression bodies forming the implementation of the `onCheckedChanged()` function. That function gets passed two parameters:

- The `CompoundButton` whose checked state changed, and
- A `Boolean` reflecting whether the new state is checked or unchecked

In our case, we just use that `Boolean` to choose which of two string resources to pass to `log()`, using the ternary operator in Java and an `if` expression in Kotlin.

Hey, You Have a Typo in `android:id`!

You may have noticed that the `android:id` value for the `Switch` is `swytch`. Considering that the `ImageButton` is `button`, you might expect that the `Switch` would be named `switch`.

However, this does not work. You cannot create a widget whose ID matches a Java keyword. `switch` is a Java keyword, so we cannot use `switch` as a widget ID. Similarly, we cannot have widgets with an ID of `if`, `else`, or `return`.

CheckBox

If you would prefer something that looks more like a classic checkbox, the Android SDK has `CheckBox`. It too is a subclass of `CompoundButton` and can be used interchangeably with `Switch`.

Our sample app has a `CheckBox` named `checkbox`:

```
<CheckBox
    android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:text="@string/checkbox_caption"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/swytch" />
```

INTEGRATING COMMON FORM WIDGETS

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

This appears beneath the Switch:

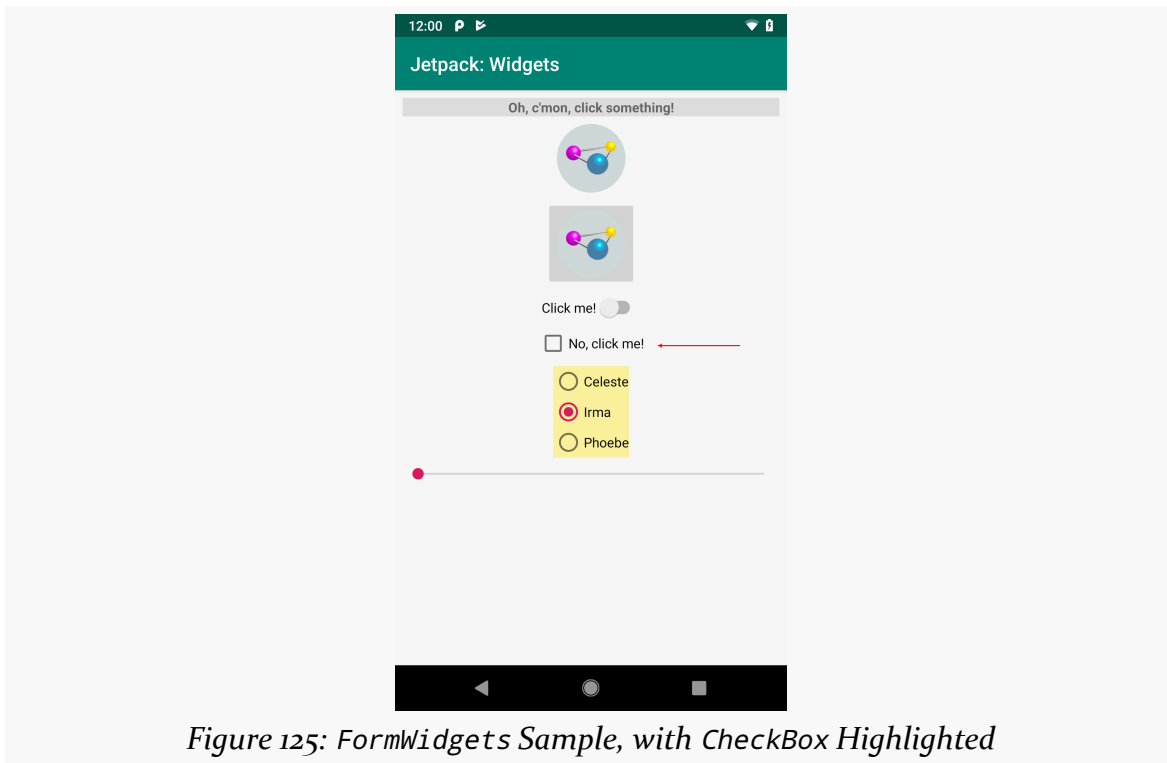


Figure 125: FormWidgets Sample, with CheckBox Highlighted

INTEGRATING COMMON FORM WIDGETS

Once again, in our case it is unchecked by default, but the user can tap on it to check it:

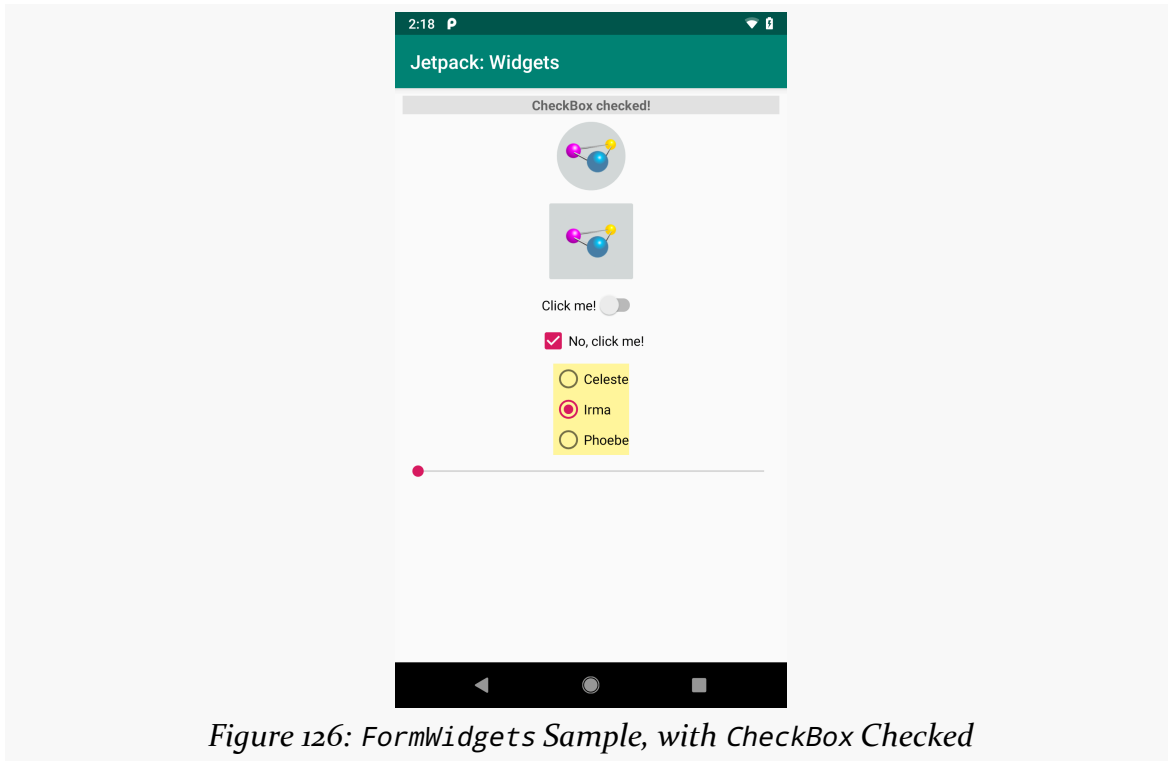


Figure 126: FormWidgets Sample, with CheckBox Checked

Android Studio Graphical Layout Editor

The CheckBox widget can be found in the “Buttons” portion of the Palette in the Android Studio Graphical Layout editor:

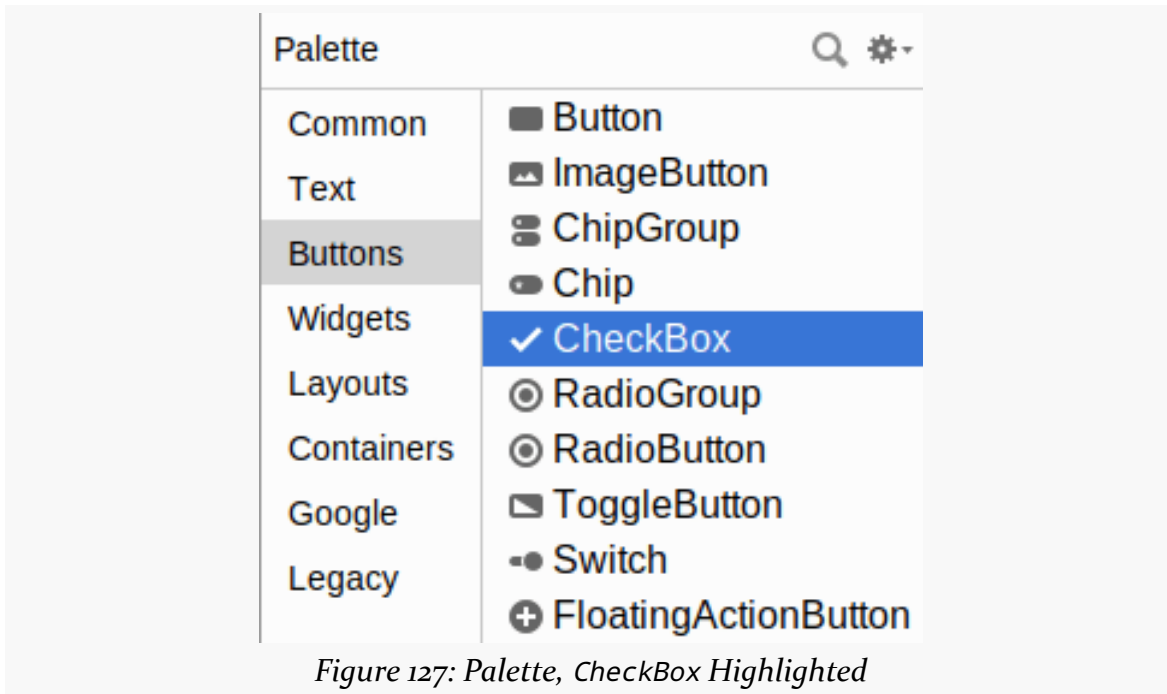


Figure 127: Palette, CheckBox Highlighted

You can drag one 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.

Reacting to Events

As with Switch, you can register a `CompoundButton.OnCheckedChangeListener` on a CheckBox using `setOnCheckedChangeListener()`:

```
binding.checkbox.setOnCheckedChangeListener((v, isChecked) ->
    log(isChecked ? R.string.checkbox_checked : R.string.checkbox_unchecked));
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

```
binding.checkbox.setOnCheckedChangeListener { _, isChecked ->
    log(if (isChecked) R.string.checkbox_checked else R.string.checkbox_unchecked)
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](#))

RadioButton and RadioGroup

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. `RadioButton` is another form of `CompoundButton` and, on its own, works like `Switch` and `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 layout resource, you can access the group from your Java/Kotlin 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 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`. Alternatively, you can use the `android:checked` attribute on one of the `RadioButton` widgets in the layout file.

Our sample app has a `RadioGroup` named `radioGroup`, containing three `RadioButton` widgets:

```
<RadioGroup
    android:id="@+id/radioGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:background="@color/radiogroup_background"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/checkbox">

    <RadioButton
        android:id="@+id/radioButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radiobutton1_caption" />
```

INTEGRATING COMMON FORM WIDGETS

```
<RadioButton
    android:id="@+id/radioButton2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:checked="true"
    android:text="@string/radiobutton2_caption" />

<RadioButton
    android:id="@+id/radioButton3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/radiobutton3_caption" />
</RadioGroup>
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

They appear beneath the CheckBox:

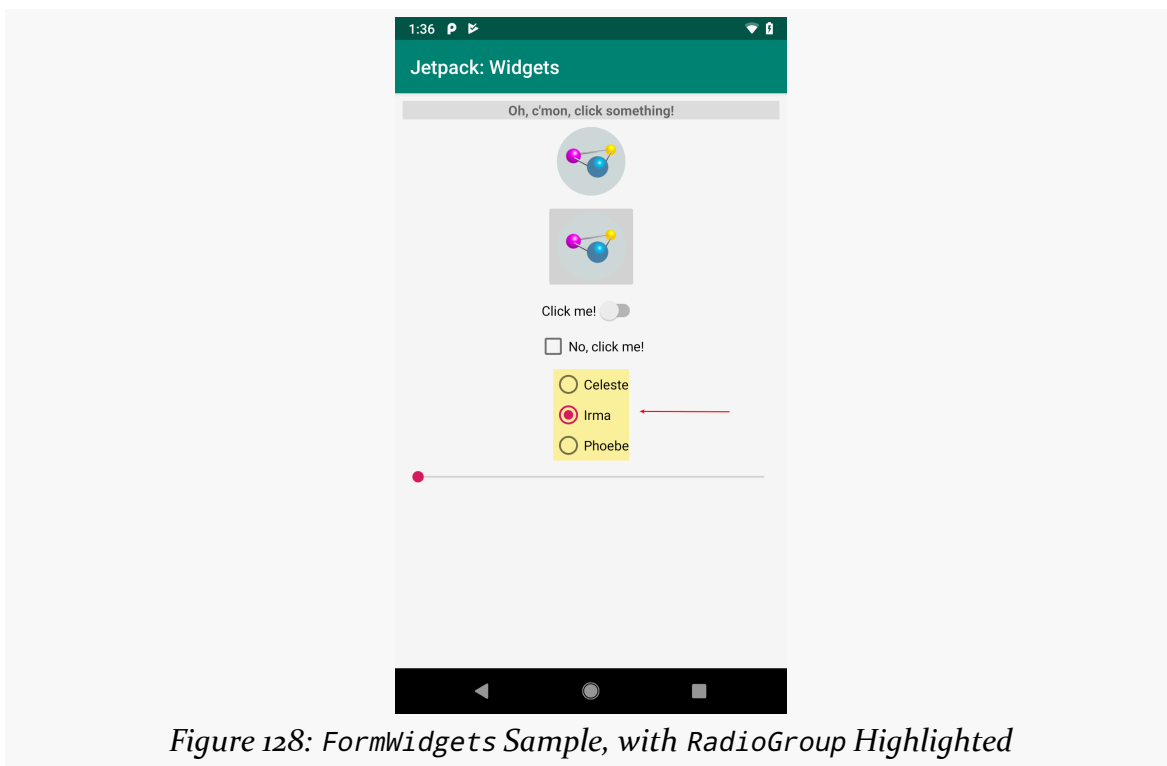


Figure 128: FormWidgets Sample, with RadioGroup Highlighted

The RadioGroup is given a pale yellow background via `android:background="@color/radiogroup_background"` so you can better see its boundaries. The three RadioButton widgets differ in ID, caption, and whether they are checked (the middle one has `android:checked="true"` to pre-select it).

`RadioGroup` inherits from `LinearLayout`, which lays its children out one after the next. By default, that is in a column, but you can use `android:orientation="horizontal"` to make them form a row.

Android Studio Graphical Layout Editor

The `RadioGroup` container and `RadioButton` widget can be found in the “Buttons” portion of the Palette in the Android Studio Graphical Layout editor:

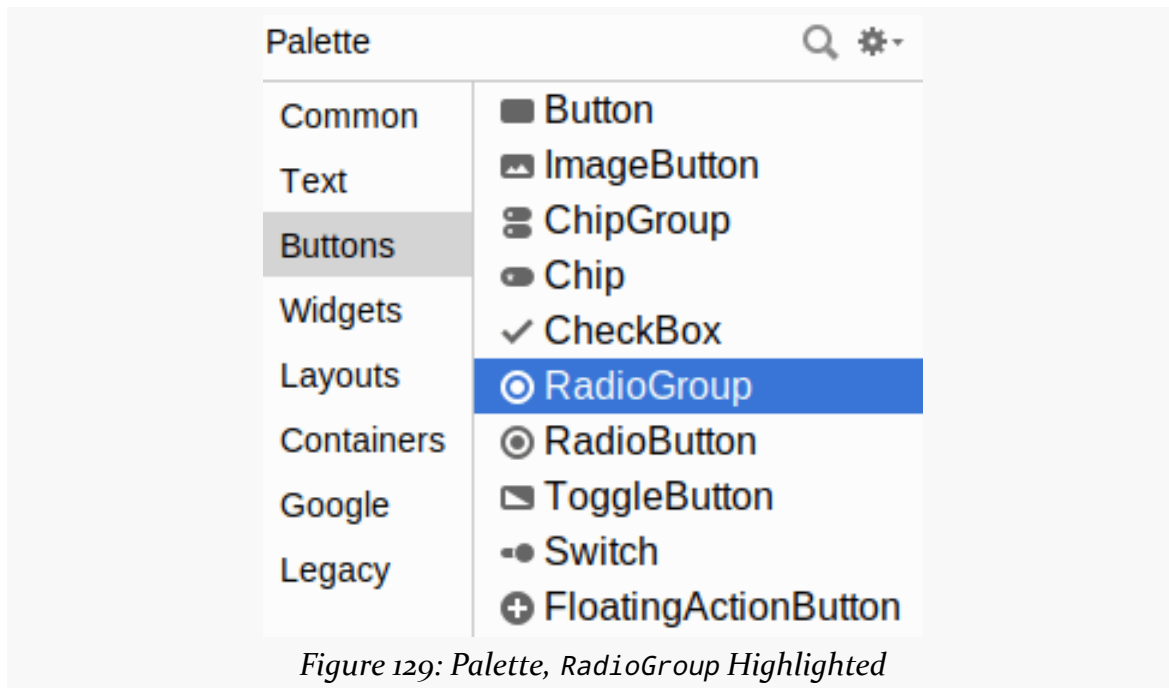


Figure 129: Palette, RadioGroup Highlighted

Dragging a `RadioGroup` into the preview gives you a box into which you can drag other widgets, such as the `RadioButton`.

Reacting to Events

It is possible to register the same sort of `CompoundButton.OnCheckedChangeListener` objects on `RadioButton`, as it too is a `CompoundButton`. More often, though, it is simpler to react to events on the `RadioGroup`, as you only need to register one listener, rather than one per `RadioButton`.

`RadioGroup` also has a `setOnCheckedChangeListener()` function. However, it takes a `RadioGroup.OnCheckedChangeListener` implementation, which is slightly different from `CompoundButton.OnCheckedChangeListener`. Your `onCheckedChanged()`

function is passed:

- The `RadioGroup` that the user interacted with, and
- The widget ID of the `RadioButton` that changed

You are *not* given a `Boolean` for the new state of that `RadioButton`, though. That is because you are always called for the newly-checked `RadioButton`, and so it is always checked. Your `onCheckedChangeListener()` function is not called for the `RadioButton` that may have been unchecked as a result of the user checking another `RadioButton`.

In the `SamplerJ` edition of `MainActivity`, we use a Java method reference to have an `onRadioGroupChange()` function on `MainActivity` be called whenever the user checks a `RadioButton` in the `RadioGroup`:

```
binding.radioGroup.setOnCheckedChangeListener(this::onRadioGroupChange);
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

There, we examine the ID of the newly-checked `RadioButton` and `log()` a different string resource for each:

```
private void onRadioGroupChange(RadioGroup group, int checkedId) {
    @StringRes int msg;

    if (checkedId == R.id.radioButton1) {
        msg = R.string.radioButton1_checked;
    }
    else if (checkedId == R.id.radioButton2) {
        msg = R.string.radioButton2_checked;
    }
    else {
        msg = R.string.radioButton3_checked;
    }

    log(msg);
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

The Kotlin code in the `Sampler` edition of `MainActivity` uses a `when` expression to `log()` the desired string resource:

```
binding.radioGroup.setOnCheckedChangeListener { _, checkedId ->
    log(
        when (checkedId) {
```

INTEGRATING COMMON FORM WIDGETS

```
R.id.radioButton1 -> R.string.radioButton1_checked
R.id.radioButton2 -> R.string.radioButton2_checked
else -> R.string.radioButton3_checked
    }
)
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](https://github.com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt))

SeekBar

Sometimes, you want the user to pick a number along some range, such as a percentage from 0% to 100%. You could use an `EditText` and then have data validation to handle illegal entries (e.g., numbers outside of your desired range).

Or, you can use a `SeekBar`.

`SeekBar` allows the user to slide a “thumb” along a bar, where different thumb positions represent different values along a range from 0 to a maximum value that you specify (the default maximum is 100).

The bottom of our layout is a `SeekBar`:

```
<SeekBar
    android:id="@+id/seekbar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/radioGroup" />
```

INTEGRATING COMMON FORM WIDGETS

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

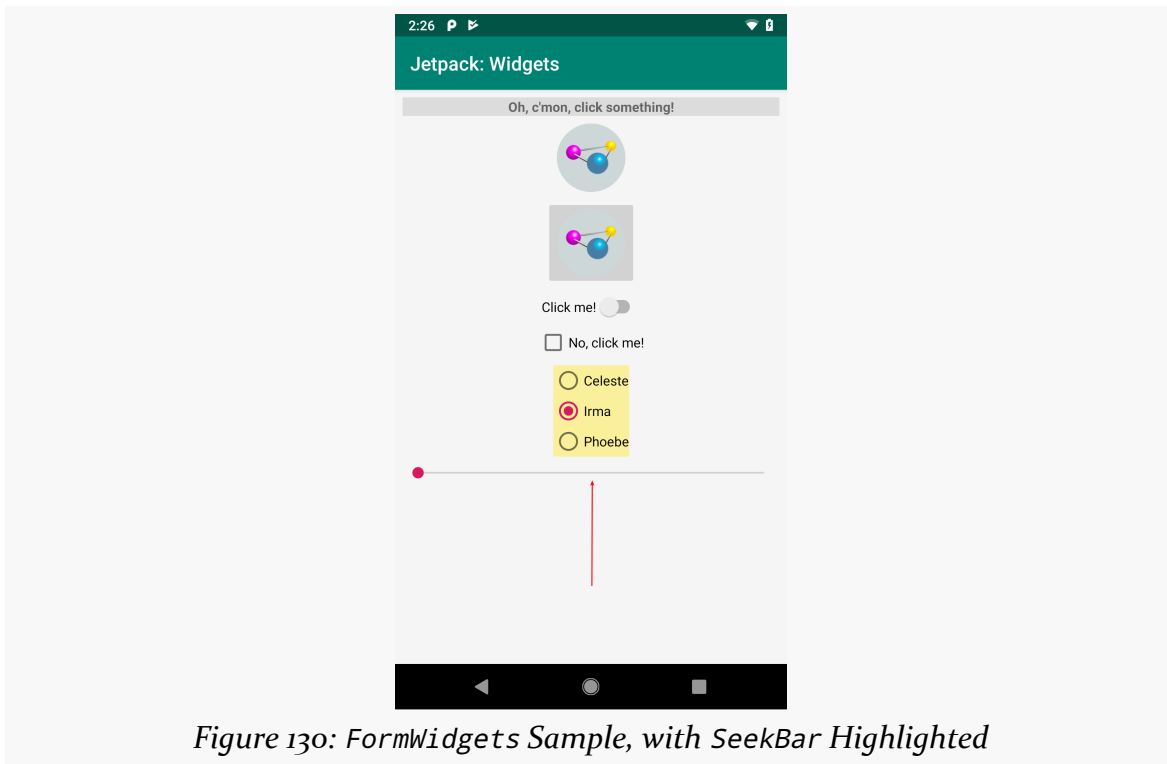


Figure 130: FormWidgets Sample, with SeekBar Highlighted

SeekBar does not inherit from `TextView`, so it has no caption. Most likely, you will want to use an adjacent `TextView` to label the `SeekBar`, so that the user has an idea of what this value represents.

Android Studio Graphical Layout Editor

The SeekBar widget can be found in the “Widgets” portion of the Palette in the Android Studio Graphical Layout editor:

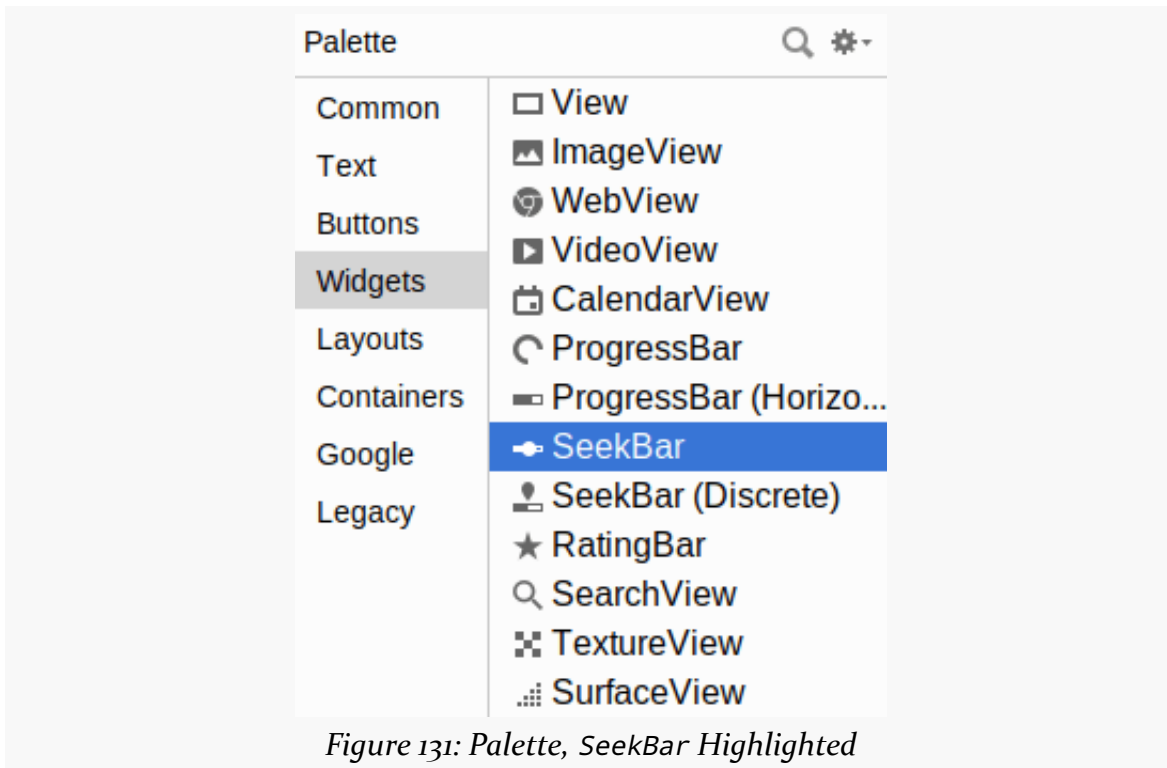


Figure 131: Palette, SeekBar Highlighted

You will notice that there is also an entry for “SeekBar (Discrete)”, where a “discrete” SeekBar shows tick marks for the values in the range. This can be useful if the range is relatively short (e.g., 0-5). To create a discrete SeekBar, you need to supply a drawable resource that provides the “tick mark”, showing the user where the thumb will snap to. If you drag a “SeekBar (Discrete)” into your layout resource, you get a SeekBar with an attribute of `style="@style/Widget.AppCompat.SeekBar.Discrete"`. This sets a particular “style” on the SeekBar, and this style sets the tick mark drawable. We will explore styles in much greater detail [later in the book](#).

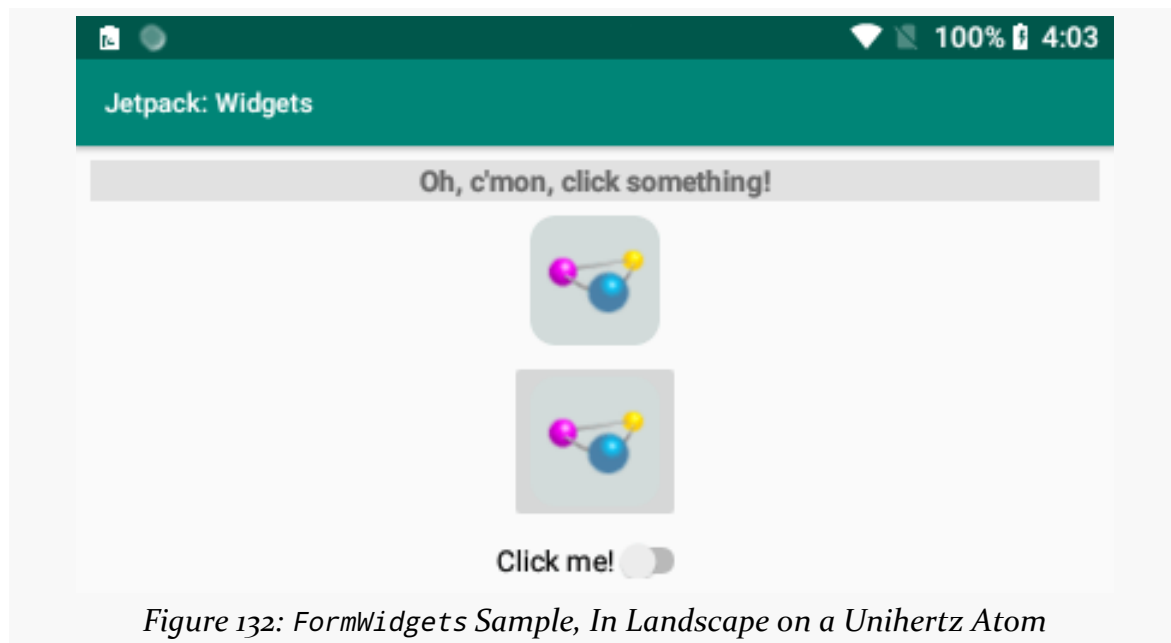
Reacting to Events

SeekBar has `setOnSeekBarChangeListener()`, which tells you about changes in the SeekBar value. However, it also requires you to override a couple of other methods,

and so it is a bit more complicated to use than are the other listeners that we have seen in this chapter. We will examine the `SeekBar.OnSeekBarChangeListener` more [later in this chapter](#) and see how it works.

ScrollView: Making It All Fit

Android devices come in all sizes. Some have very large screens, while others have very small screens. The problem with very small screens is that your forms do not always fit the available space, particularly in landscape mode:



That is why the entire `ConstraintLayout` is wrapped in a `ScrollView`. `ScrollView` provides a vertically-scrolling area for your form. The user can swipe up and down to pan around the form and see all of it.

`ScrollView` itself is a very simple container class, holding exactly one child (in this case, the `ConstraintLayout`), and making it scrollable. So, our entire `activity_main` layout is:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView 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:padding="@dimen/container_padding">

<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/log"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:background="@color/log_background"
        android:gravity="center_horizontal"
        android:text="@string/log_default"
        android:textStyle="bold"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <ImageView
        android:id="@+id/icon"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_row"
        android:contentDescription="@string/icon_caption"
        android:src="@mipmap/ic_launcher"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/log" />

    <ImageButton
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_row"
        android:contentDescription="@string/button_caption"
        android:src="@mipmap/ic_launcher"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/icon" />

    <Switch
        android:id="@+id/swytch"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_row"
        android:text="@string/switch_caption"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
```

```
app:layout_constraintTop_toBottomOf="@id/button" />

<CheckBox
    android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:text="@string/checkbox_caption"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/switch" />

<RadioGroup
    android:id="@+id/radioGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:background="@color/radiogroup_background"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/checkbox">

    <RadioButton
        android:id="@+id/radioButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radioButton1_caption" />

    <RadioButton
        android:id="@+id/radioButton2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="@string/radioButton2_caption" />

    <RadioButton
        android:id="@+id/radioButton3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radioButton3_caption" />
</RadioGroup>

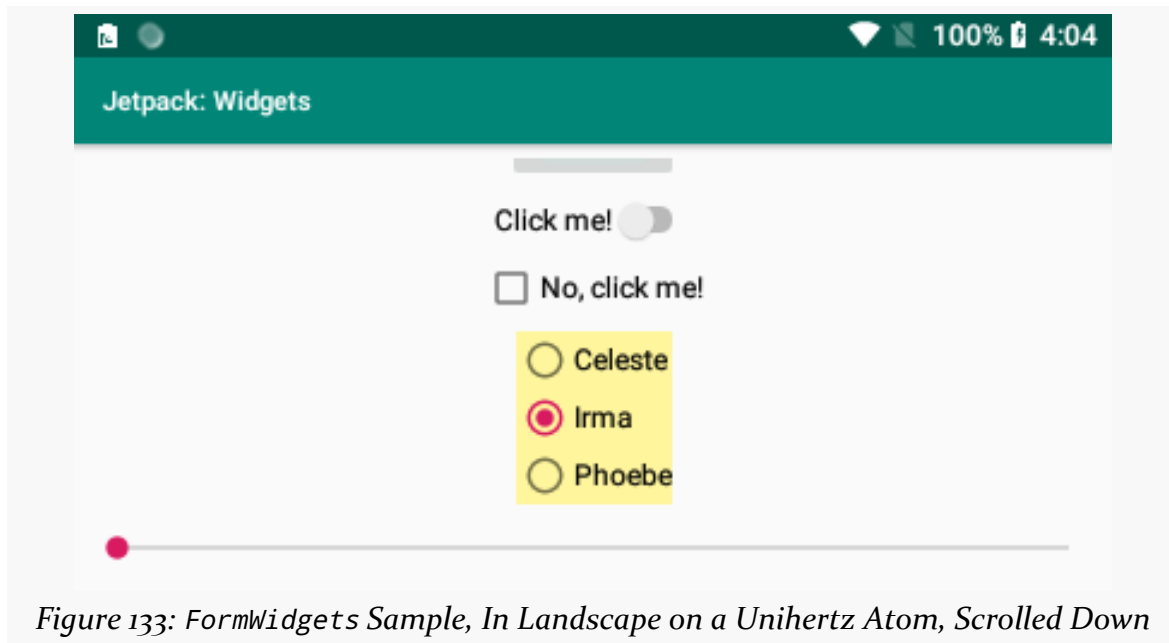
<SeekBar
    android:id="@+id/seekbar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    app:layout_constraintEnd_toEndOf="parent"
```

INTEGRATING COMMON FORM WIDGETS

```
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toBottomOf="@id/radioGroup" />

</androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))



Note that there is also a `HorizontalScrollView` that will allow your form to be wider than the screen, whereas `ScrollView` allows your form to be taller than the screen.

Other Notes About the Sample

There are a few more interesting elements of the sample app that are worth noting.

`android:gravity`

There is a `TextView` at the top of the layout. Initially, it shows the message “Oh, c’mon, click something!”, but if you follow those instructions and start interacting with the other widgets, the message changes to reflect the recent event (e.g., “Button clicked!”).

The `TextView` has an `android:gravity="center_horizontal"` attribute:

```
<TextView
    android:id="@+id/log"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:background="@color/log_background"
    android:gravity="center_horizontal"
    android:text="@string/log_default"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

“Gravity” means “hey, if there is room, position my content here”. `center_horizontal` gravity says “hey, if there is room horizontally, center my content”. For a `TextView`, the “content” is the text. The width is set to `0dp`, which for a child of `ConstraintLayout` means that the `TextView` will be stretched between its start and end anchor points. Those are at the edges of the `ConstraintLayout`, so the `TextView` stretches to fill the width of the screen, less `8dp` of padding that we have in the `ScrollView`. The `TextView` has a gray background, so in the screenshots, you see the full extent of the `TextView` plus the centered message.

Another way of centering things is through `ConstraintLayout` itself. If we switch the width of the `TextView` to `wrap_content`, `ConstraintLayout` will interpret that as meaning that the `TextView` should be centered between its start and end anchor points (assuming there is no bias setting to slide it one way or another):

```
<TextView
    android:id="@+id/log"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@color/log_background"
    android:text="@string/log_default"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

INTEGRATING COMMON FORM WIDGETS

However, in that case, the `TextView` itself is only as big as its content, so the gray background does not stretch to fill the screen:

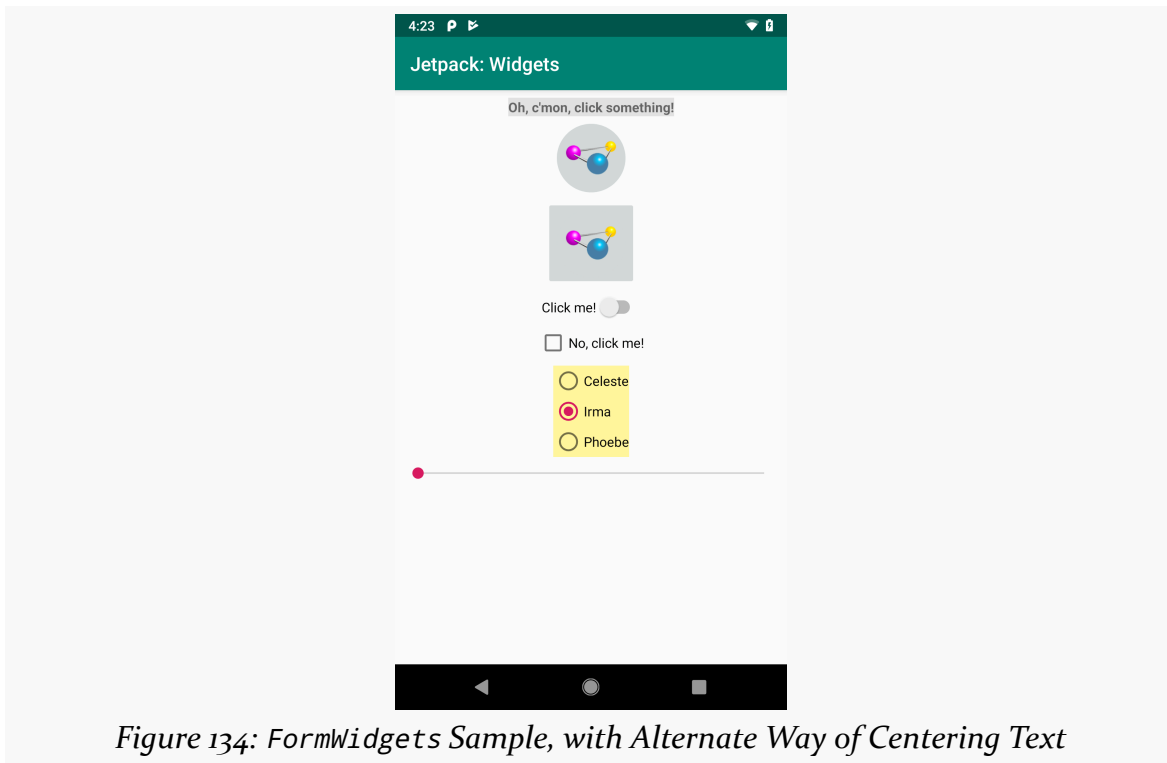


Figure 134: FormWidgets Sample, with Alternate Way of Centering Text

`log()`

That `TextView` is the `log` widget. Our `log()` function is set up to:

- Display a message in that `TextView`, and
- Log that same message to Logcat

So, we have a `log()` method in Java:

```
private void log(@StringRes int msg) {  
    binding.log.setText(msg);  
    Log.d(TAG, getString(msg));  
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](https://github.com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java))

...and a corresponding `log()` function in Kotlin:

INTEGRATING COMMON FORM WIDGETS

```
private fun log(@StringRes msg: Int) {  
    binding.log.setText(msg)  
    Log.d(TAG, getString(msg))  
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](#))

The parameter to `log()` has a `@StringRes` annotation. This has no effect at runtime. However, it will help the build tools avoid some compile-time problems akin to the bug from [the chapter on debugging](#). There, we passed an arbitrary `Int` to `setText()` on a `TextView`. That function expects a string resource, and so it crashes if you pass some `Int` that does not represent a string resource. The `@StringRes` annotation on an `Int` says “hey! this `Int` should point to a string resource!”. If we attempt to call `log()` with an arbitrary `Int`, we will get an error in Android Studio:

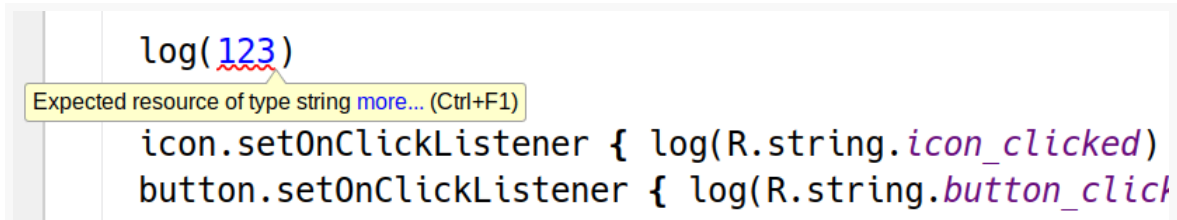


Figure 135: Android Studio, Showing Lint Error

This error message is being provided by a tool called “Lint”. The IDE uses Lint and its set of rules to identify possible problem spots in the code, such as calling `log()` with a value that is not a string resource ID. Consider it a reminder service, hinting about some possible flaws in your code.

INTEGRATING COMMON FORM WIDGETS

We also want to log messages from our SeekBar. There, though, we want to show the current value of the SeekBar, based on where the user has positioned the thumb:

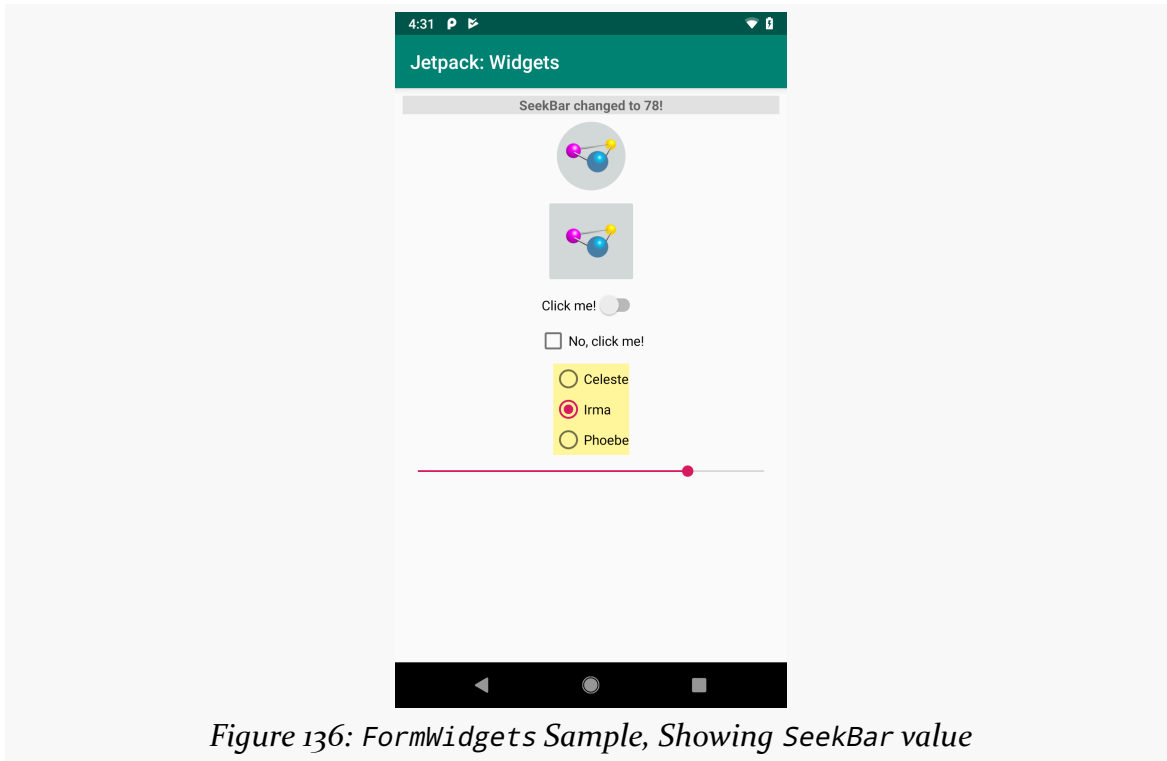


Figure 136: FormWidgets Sample, Showing SeekBar value

Our `log()` function is not set up for that, so we instead just do that bit of logging directly from our `OnSeekBarChangeListener`, whether that is implemented in Java:

```
binding.seekbar.setOnSeekBarChangeListener(  
    new SeekBar.OnSeekBarChangeListener() {  
        @Override  
        public void onProgressChanged(SeekBar seekBar, int progress,  
                                       boolean fromUser) {  
            String msg = getString(R.string.seekbar_changed, progress);  
  
            binding.log.setText(msg);  
            Log.d(TAG, msg);  
        }  
  
        @Override  
        public void onStartTrackingTouch(SeekBar seekBar) {  
            // ignored  
        }  
    })
```

INTEGRATING COMMON FORM WIDGETS

```
@Override
public void onStopTrackingTouch(SeekBar seekBar) {
    // ignored
}
});
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

...or Kotlin:

```
binding.seekbar.setOnSeekBarChangeListener(object :
    SeekBar.OnSeekBarChangeListener {
    override fun onProgressChanged(
        seekBar: SeekBar,
        progress: Int,
        fromUser: Boolean
    ) {
        val msg = getString(R.string.seekbar_changed, progress)

        binding.log.text = msg
        Log.d(TAG, msg)
    }

    override fun onStartTrackingTouch(seekBar: SeekBar) {
        // unused
    }

    override fun onStopTrackingTouch(seekBar: SeekBar) {
        // unused
    }
})
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](#))

Our string resource has a placeholder in it:

```
<string name="seekbar_changed">SeekBar changed to %d!</string>
```

(from [FormWidgets/src/main/res/values/strings.xml](#))

Here, %d in our string resource means “we will supply some Int at runtime to fill in here”. `getString()` not only takes a string resource ID, but it optionally takes objects to use for those placeholders.

The primary function of an `OnSeekBarChangeListener` is `onProgressChanged()`. This will be called when the `SeekBar` value is adjusted. You are passed:

- The Seekbar itself,
- The new value of the SeekBar, and
- A Boolean indicating if the user changed the SeekBar value or if you did programmatically

So, our `getString(R.string.seekbar_changed, progress)` call takes the progress value and uses that to replace the `%d` in the string resource, giving us the text to display in the `TextView` and `Logcat`.

`OnSeekBarChangeListener` also has two other methods:

- `onStartTrackingTouch()`, which will be called when the user starts dragging the thumb
- `onStopTrackingTouch()`, which will be called when the user stops dragging the thumb

Most uses of `SeekBar` do not need to worry about those events, but we have to have those methods to satisfy the compiler, so the `FormWidgets` sample just has them as empty functions.

Contemplating Contexts

When we call `getString()` to convert a string resource into its actual `String` value, we have been calling it on our `MainActivity`. However, in reality, that function is implemented on `Context`. `MainActivity` extends `AppCompatActivity`, which inherits from `Activity`, which implements the `Context` interface.

We use `Context` objects a *lot* in Android.

So, in this chapter, let's quickly review what a `Context` is, where to get one, and how we use it.

It's Not an OMG Object, But It's Close

Back when Android was first created, in the mid-2000's, the dominant form of Java programming was for server-side Web development, using servlets, WARs, and so on. In Web development, there is often some form of "session" object. At minimum, this object represents the user's session with the server and represents a place where the servlet can stash values from one request that might be used in a subsequent request from the same user.

Computer programming has lots of "anti-patterns": things that we really should not do. One of those anti-patterns is ["the God object"](#), where we have one class that tries to do far too much. Session objects in Web development have a tendency to become God objects, because it is simple to say, "oh, well, we'll just have the session handle that", even for things that have little to do with maintaining user state across requests.

In Android, the God object is `Context`. A lot of functionality routes through a `Context`, either because:

- Our only ways of accomplishing a particular thing involve functions on a Context, or
- Our only way to get at some other object that accomplishes that thing is by calling a function on the Context

For lightweight app development, such as most of this book's samples, this does not pose much of a problem. Any code in an activity, for example, has easy access to a Context, as the activity itself *is* a Context. In larger apps, though, quite a bit of time is spent trying to figure out how best to write the code to reduce the number of places where we need a Context.

The Major Types of Context

Context is an interface, so in theory anything could be a Context. However, there are only a few Context implementations that actually “do the heavy lifting” required of a fully-working Context.

Context from Components

Primarily, your components will give you a Context.

The one component that we have worked with so far is Activity. An Activity is itself a Context, which is why we can call `getString()` on it.

There are three other types of components:

- Service, which, like Activity, implements Context itself
- BroadcastReceiver, where you get a Context via its `onReceive()` function that you implement or inherit
- ContentProvider, where you get a Context by calling `getContext()` when you need one

The *vast* majority of the time, using a Context that you get from a “nearby” component is the right answer. For example, for code that is in an Activity, most of the time you will want to use the Activity itself as your Context.

Application

There is one other “root” Context besides your components: Application.

Each Android app process has an `Application` singleton. This object is created when the process is created, and it lives through the life of that process. In your regular app code, you get access to this singleton... by calling `getApplicationContext()` on some other `Context`.

As a result, `Application` may seem somewhat pointless. If you already have a `Context`, why would you need `getApplicationContext()` to get a *different* `Context`?

The answer lies in lifespan. Other types of `Context` — particularly an `Activity` — have short lifespans. As we will see [later in the book](#), an `Activity` will come and go from the screen, even in some cases where the user might not think that there has been any change. Sometimes, we will need a `Context` that lives longer than this, and for that, we have `Application`.

That being said, [only use `Application` when you have a clear reason why you need it](#).

Instrumented Tests

Note that in testing we get a `Context` in other ways.

For example, back in [the chapter introducing testing](#), we had an `ExampleInstrumentedTest` class, both in Kotlin:

```
package com.commonware.jetpack.hello

import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
    }
}
```

```
    val appContext = InstrumentationRegistry.getInstrumentation().targetContext
    assertEquals("com.commonware.jetpack.hello", appContext.packageName)
  }
}
```

...and in Java:

```
package com.commonware.jetpack.hello;

import android.content.Context;
import androidx.test.platform.app.InstrumentationRegistry;
import androidx.test.ext.junit.runners.AndroidJUnit4;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

/**
 * Instrumented test, which will execute on an Android device.
 *
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
 */
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Test
    public void useAppContext() {
        // Context of the app under test.
        Context appContext =
            InstrumentationRegistry.getInstrumentation().getTargetContext();
        assertEquals("com.commonware.jetpack.hello", appContext.getPackageName());
    }
}
```

`InstrumentationRegistry.getTargetContext()` returns a `Context` from the app that we are testing.

Key Context Features

Many things take a `Context`. Constructors for all of our widgets and containers, for example, take a `Context` as a parameter.

In terms of directly calling functions on a `Context`, though, there are a few categories into which those functions fall.

Access to Resources and Assets

If you want to get the value or otherwise use a resource, you will wind up using a `Context`.

Sometimes, that comes from passing the `Context` to something else as a parameter. We will see examples of that later in the book, such as [data binding](#).

Sometimes, that comes from calling convenience functions on `Context` itself, such as `getString()`.

In general, though, access to resources comes from a `Resources` object, and you get one of those by calling `getResources()` on a `Context`. `Resources` has functions to retrieve most types of resources: strings, dimensions, colors, and so on. The convenience functions on `Context` simply delegate to the `Resources` object — for example, `getString()` on `Context` just forwards your request to `getString()` on `Resources`.

In addition to resources, an Android app can also have assets, found in an `assets/` directory that sits alongside directories like `src/` and `res/`. If you package assets this way with your app, calling `getAssets()` on a `Context` gives you an `AssetManager`. From there, you can work with the assets, such as by calling `open()` to get an `InputStream` from which you can read an asset's content.

Access to Key Directories

`Resources` and assets are packaged with the app. Beyond those, we have the ability to write to files on the local filesystem and read them back later. On the whole, this is standard I/O using Java classes like `File`, `InputStream`, `OutputWriter`, and so on. The big limitation is *where* we can read and write.

For that, `Context` has a family of functions, like `getFilesDir()`, that return `File` objects representing directories that we can read from and (usually) write to. We will explore those functions in detail [later in the book](#).

Access to System Services

Lots of stuff used by your app is actually managed outside of your app.

For example, suppose that you want to find out the device's location. Your app code does not work with GPS hardware directly. Instead, it will work with a

LocationManager or something that wraps around a LocationManager. LocationManager is a “system service”, and it in turn will talk to other parts of the OS that, eventually, work with the GPS hardware.

Whenever you need access to a “system service”, the typical approach to get one is to call `getSystemService()` on a Context. Occasionally, there will be other approaches:

- Some system services have convenience functions on Context, such as `getPackageManager()` to get a PackageManager instance
- Some system services have a function that you can use to get an instance by way of a Context, such as `LayoutInflater.from()`

Access to Other Components

As we start to explore Android’s components more, you will see that if we want to work with other components from an existing component, we will need a Context.

Specifically, we use a Context to:

- Start an Activity
- Start or bind to a Service
- Send a system broadcast to be picked up by a BroadcastReceiver
- Work with a ContentProvider

Know Your Context

As your app grows, you will start having classes that are not Android components and do not have direct access to some Context. Instead, they will be talking with your Web services, or storing things in a local database, or performing any number of other tasks that are important for your app.

In an ideal world, few of those classes will need a Context. For those that do, you will need to decide *what* Context they will use:

- Do they take a Context as a function parameter for a particular operation, so the caller chooses the Context?
- Do they take a Context as a constructor parameter, so the choice of Context has to take into account how long this object is needed?
- Do they do something more elaborate, such as use [dependency inversion](#)?

Context **Anti-Patterns**

The reason why you need to know what Context you are using in different circumstances is because there are several anti-patterns when working with Context. Some of these will result in warnings in the IDE, but not always, as sometimes the IDE cannot identify that you are implementing an anti-pattern.

We will explore these anti-patterns later in the book:

- [Avoiding static references to component Context objects](#)
- [Trying to use Application for everything](#)
- [Assuming that a Context is always of some specific type](#)

Each app has an icon. For most apps, the user will see that icon in their launcher, for whatever activity (or activities) are advertised as belonging in the launcher. In addition, this icon can appear in other places, such as in the Settings app.

App 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 app icons 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

Google has been making app icons 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.)

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:

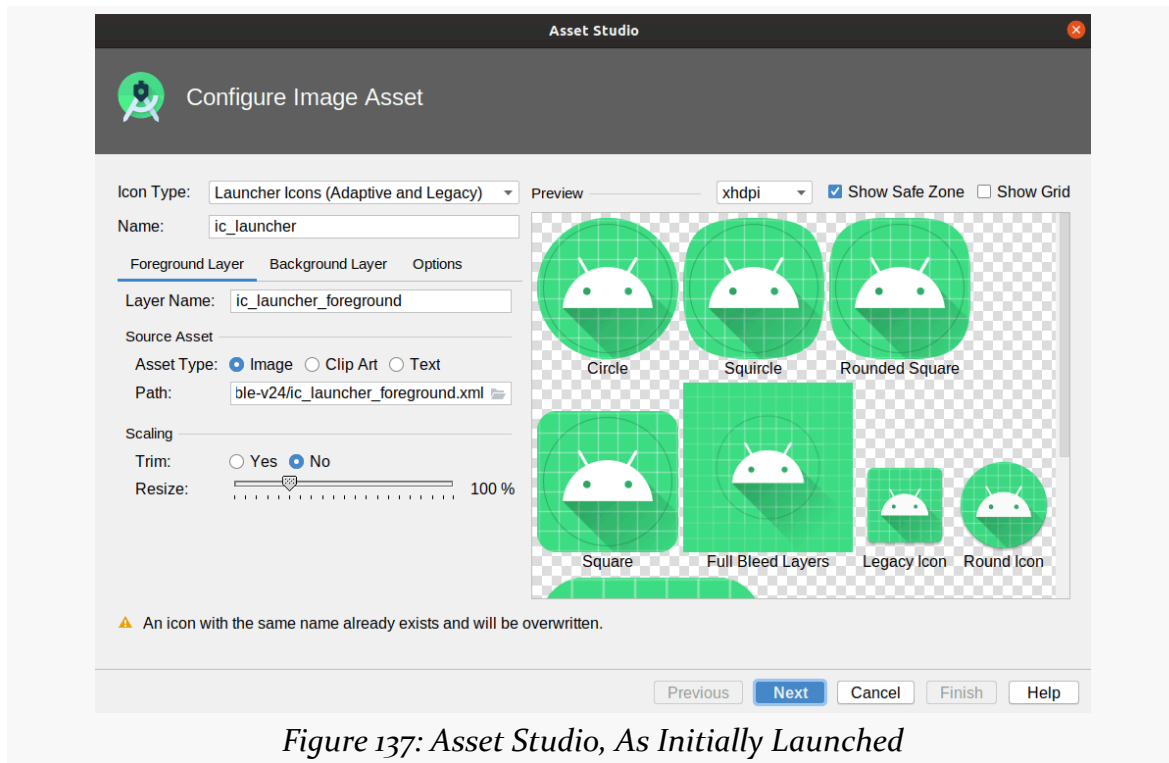


Figure 137: Asset Studio, As Initially Launched

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.

ICONS

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 folder button inside 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:

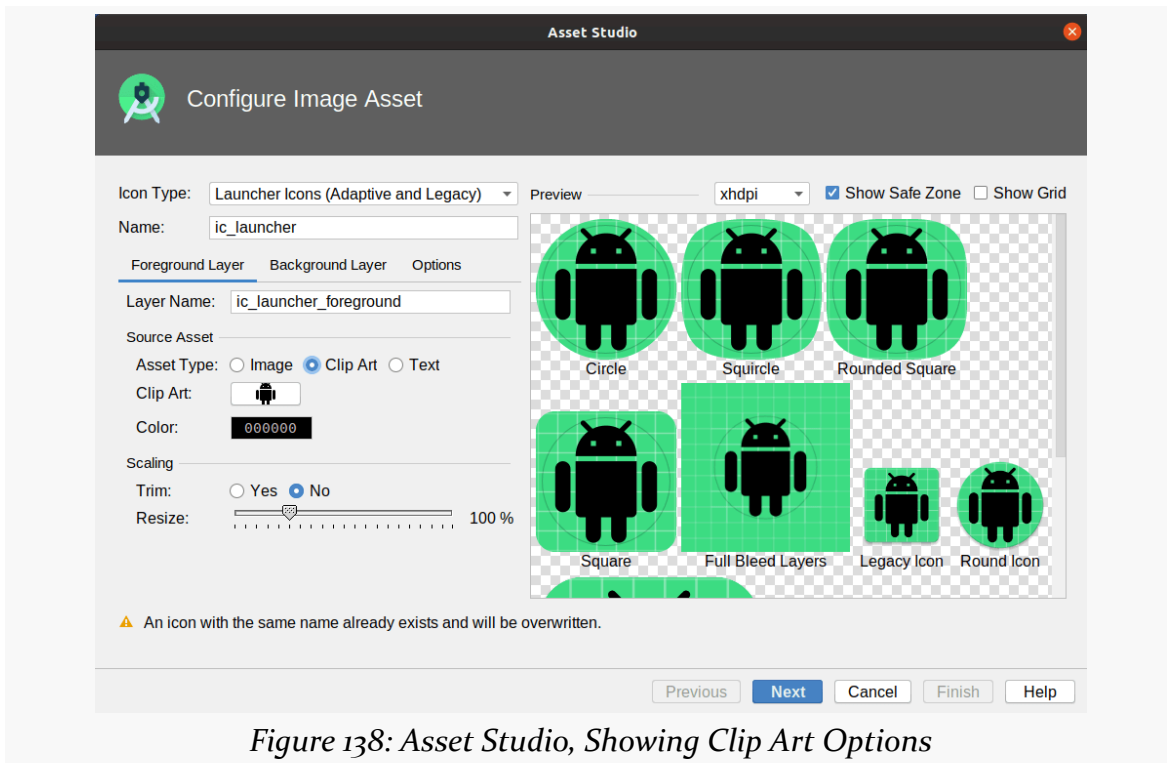


Figure 138: Asset Studio, Showing Clip Art Options

ICONS

- 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:

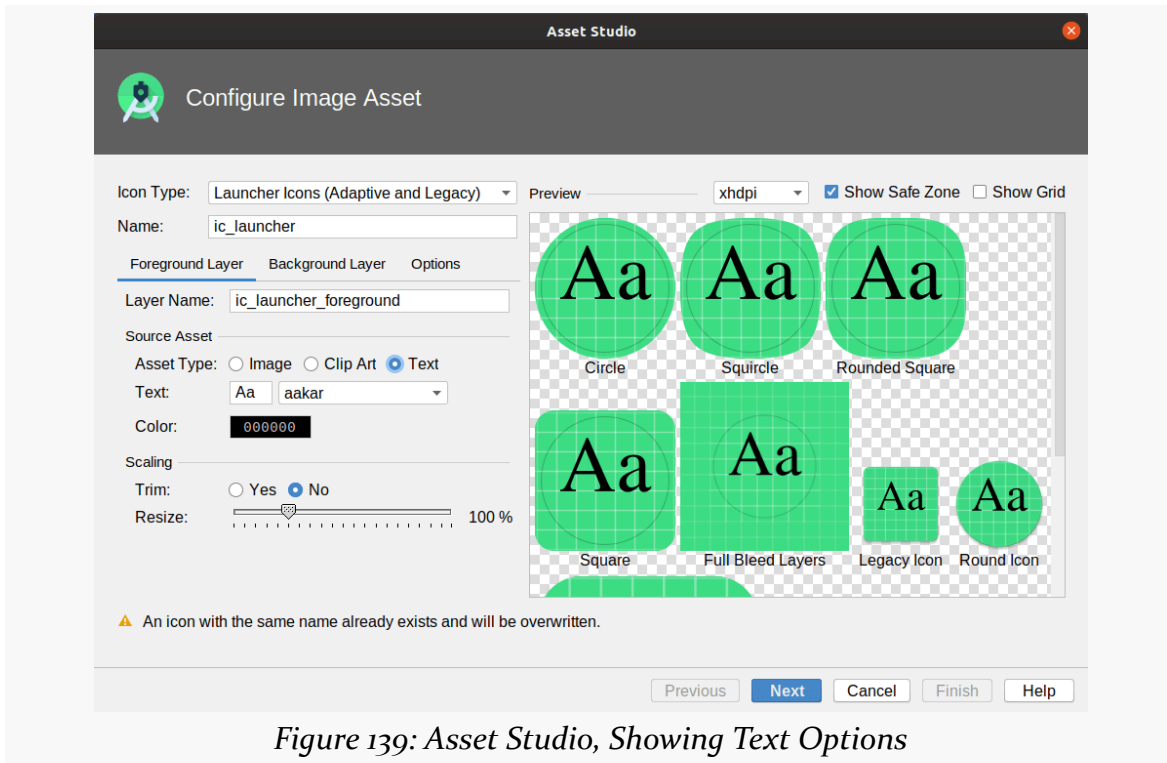


Figure 139: Asset Studio, Showing Text Options

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 anything 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 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:

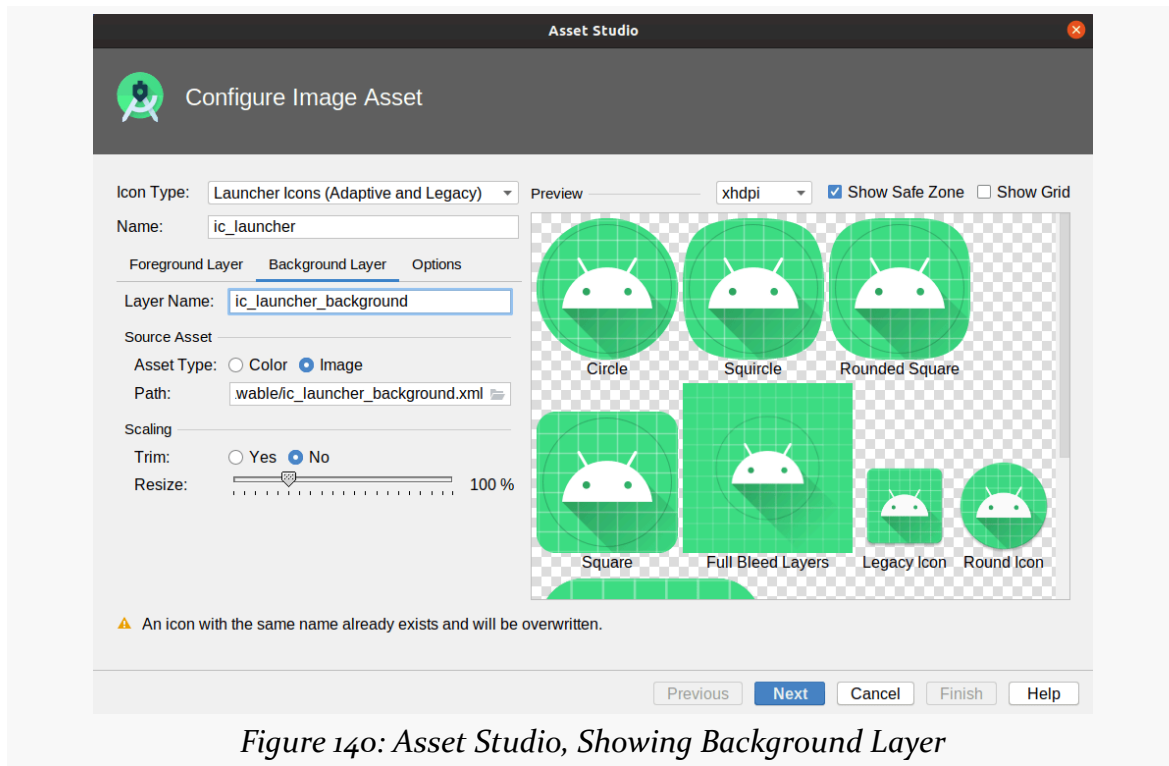


Figure 140: Asset Studio, Showing Background Layer

Your two main options are:

- a flat color
- an image of your choosing (akin to the foreground, where you click the folder button inside 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.

Options

On Android 8.0+ devices, the foreground layer, background layer, and device-chosen

ICONS

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 Options tab helps you decide what that is:

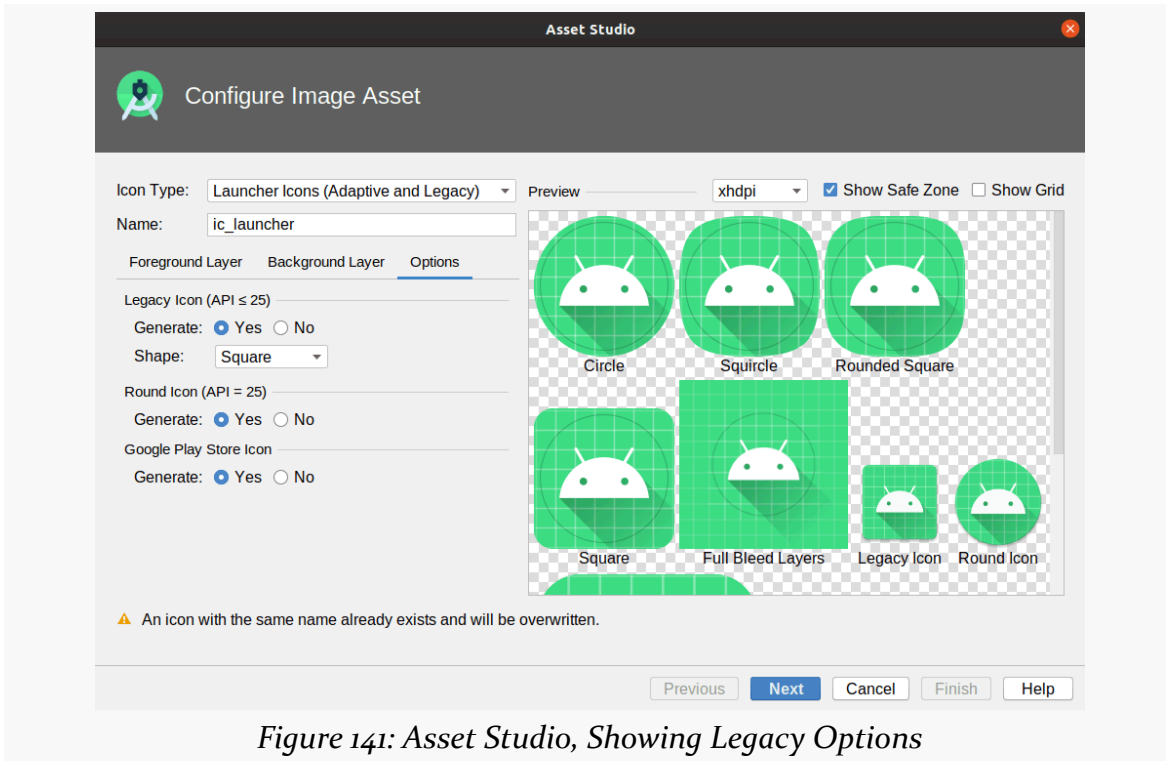


Figure 141: Asset Studio, Showing Legacy Options

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:

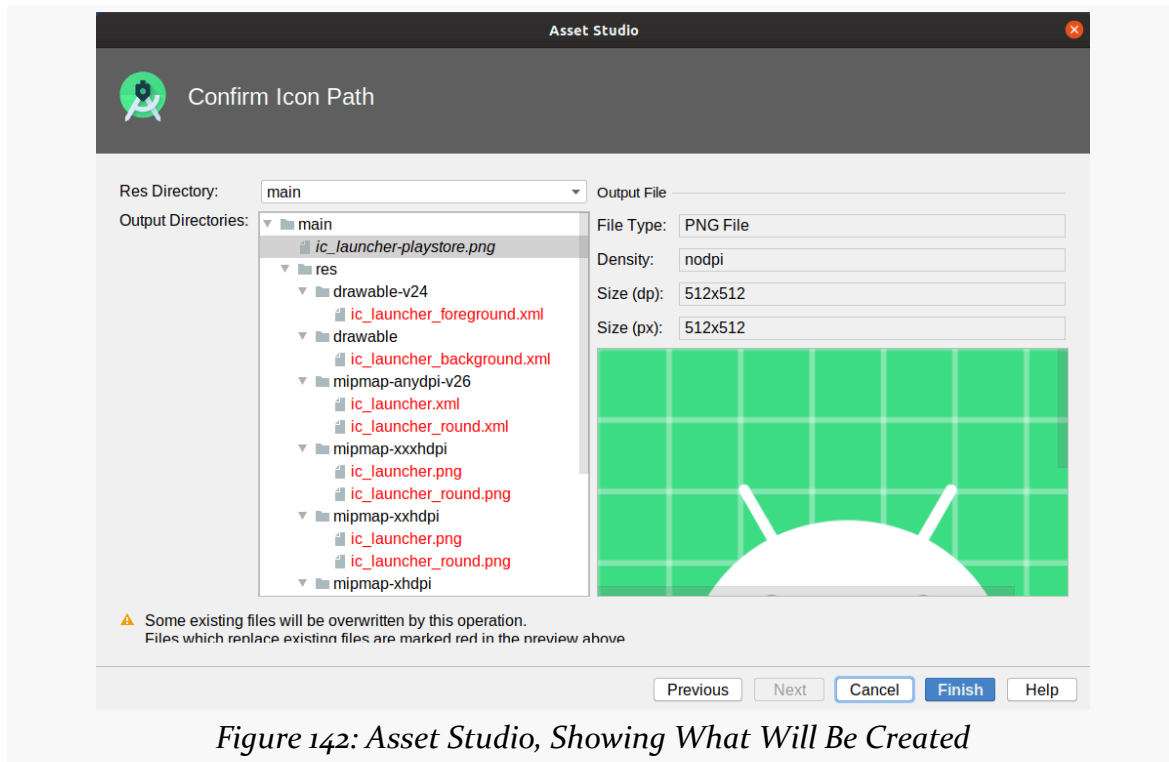


Figure 142: Asset Studio, Showing What Will Be Created

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, a concept that we will explore [in an upcoming chapter](#).

Using In Your Manifest

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

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

You can also have an `android:roundIcon` attribute. This is for the Android 7.1-specific scenario of having a dedicated round icon resource. If you elected to have Android Studio create a round icon for you, ensure that your `<application>` element has an `android:roundIcon` attribute that points to the round icon resource (e.g., `@mipmap/ic_launcher_round`).

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](#)

Adding Libraries

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 members of your development team, but far more comes from outsiders: Google and other Android developers.

You have seen some of this 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.

You have seen some of that already too, in the form of libraries for things like `ConstraintLayout`.

In this chapter, we will explore a bit more about these libraries, how you find them, and how you integrate them.

Depending on a Local JAR

Some projects that you look at will have an `implementation` statement in their module's `build.gradle` file that looks like this:

```
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. 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.

Artifacts and Repositories

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 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. And there can be private repositories, such as ones used by organizations for their own private artifacts, used by their private projects.

So, we have dependencies like these:

```
dependencies {  
  
    implementation 'androidx.core:core-ktx:1.6.0'  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'com.google.android.material:material:1.4.0'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
    testImplementation 'junit:junit:4.+'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
}
```

For those, Gradle checks with our registered repositories and says “hey, do you have this particular artifact?”. This is based on the group (e.g., `org.jetbrains.kotlin`), the artifact ID (e.g., `kotlin-stdlib`), and the desired artifact version. If a match is found, Gradle downloads it and caches it for later use. If no match is found, you get a build error pointing out the difficulty.

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 libraries that address whatever problems you need to solve.

Many libraries will be covered in this book. Yet more are covered in other CommonsWare books. A few more are in the developer documentation.

The biggest catalog of open source libraries is [the Android Arsenal](#). Here you can browse and search for libraries. Each listing will contain links to where you can find out more about that particular library, typically in the form of a GitHub repository.

Beyond that, you will find blog posts, Stack Overflow answers, and other resources pointing out candidate libraries to use.

Configure the Repositories

If you are using one of Google's libraries, a project created in Android Studio should be set up with the proper artifact repository already. Your project should also come 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, you may find some libraries that advise you to configure support for jitpack.io:

```
repositories {  
    maven {  
        url "https://jitpack.io"  
    }  
}
```

This is because they are using JitPack to publish their libraries, instead of JCenter.

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, for if you create a more complex app that involves multiple modules.

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. If the library does not state what artifact repository it uses, try adding the dependency, and if it works, then (hopefully) they published the artifact on JCenter.

Identify the Version That You Want

As noted above, each artifact has an identifier made up of three pieces: a group ID, an artifact ID within the group, and a version number.

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.

For Google’s artifacts, you can browse [a Web page for their Maven repository](#) to find the available versions for an artifact given its group ID and artifact ID.

For other artifacts, you usually go to the support site for that artifact, which for many artifacts is a GitHub project. A well-maintained library will have details of the latest version of that artifact, so you know what version to request in your module’s build.gradle file.

Add the Dependencies

Then, you just need to add the appropriate implementation lines for whatever dependencies that you wish to add, alongside the existing ones in your module’s build.gradle file.

Note that some libraries showing sample code for adding them to your build.gradle file will show a slightly different syntax, with a compile keyword instead of implementation:

```
compile 'com.whatever:something:1.2.3'
```

Older versions of Android Studio (3.0 and back) used compile where we now use implementation, and older documentation may still reference that compile keyword as a result.

We will see lots of dependencies closures throughout the rest of this book, showing different artifacts that we can depend upon.

Employing RecyclerView

If you have spent much time with Android devices and apps, no doubt that you will have seen apps that show lists or grids of stuff: contacts, songs, movies, books, to-do items, etc.

If the app was written in 2015 or later, there is a very good chance that it uses RecyclerView for those lists and grids. In this chapter, we will explore what RecyclerView is and how you can use it for your own collections of data.

Recap: Layouts vs. Adapter-Based Containers

As we saw back in [the chapter on ConstraintLayout](#), there are two major types of ViewGroup implementations:

- There are those that organize a set of children known at the time that you are writing the app
- There are those that organize a set of children that cannot be known until the app is run

Mostly, ConstraintLayout is for the first scenario. You know that you will have a certain set of widgets that you want to appear, and you need to size and position them. ConstraintLayout — and the legacy containers, like LinearLayout — excel at this role.

However, quite frequently, we have collections of stuff to display. We might know what sorts of stuff would be in the collection (contacts, songs, movies, books, to-do items, etc.). But we do not know *how big* the collection will be. Perhaps when we first run the app, the collection will be empty. Over time, the collection might grow. If we are reusing data from other places, such as the user's contact manager, there

might be lots of data from the outset. We just do not know.

ConstraintLayout would be useful for displaying an individual item out of that collection of stuff. However, trying to use ConstraintLayout and ScrollView to handle an arbitrary-sized collection would be really painful.

Fortunately, we have options.

The original solution for this sort of problem was the AdapterView family of classes, notably ListView. AdapterView worked but was fairly inflexible. RecyclerView was created as an alternative, and it is the dominant solution for this problem today. We will discuss ListView and the rest of AdapterView [towards the end of the chapter](#), but outside of certain scenarios, your focus nowadays should be on RecyclerView.

The Challenge: Memory

Suppose that our UI is supposed to look like this:

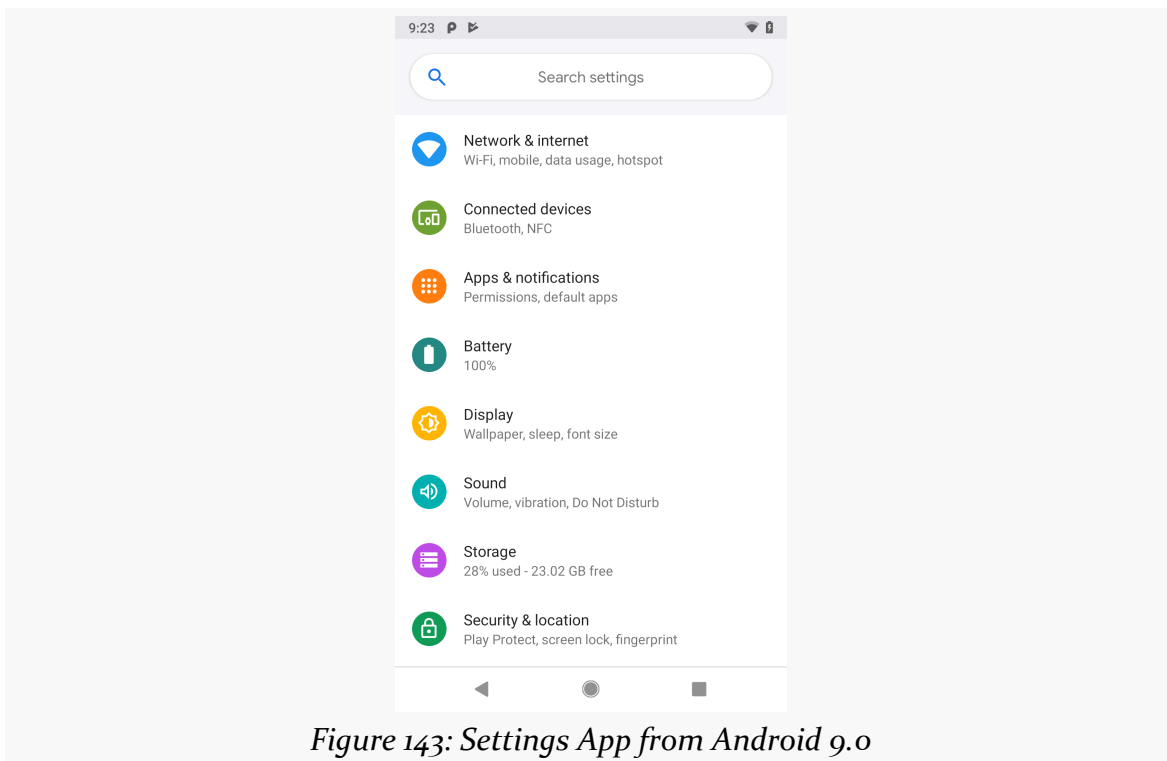


Figure 143: Settings App from Android 9.0

Here we have a vertically-scrolling list of items. Each item appears to have an

ImageView and a pair of TextView widgets.

This happens to be from the Settings app on an Android 9.0 device. If you have spent much time with Settings, you know that while it has a bunch of top-level options like these, there are only so many. We could, if needed, have one ConstraintLayout in a ScrollView that managed them.

But what happens if our list needs to have thousands of items in it?

Each widget — TextView, Button, ImageView, etc. — takes up around 1KB of memory, not counting its content (text, image, whatever). Several thousand widgets means several MB of memory.

On Android, the amount of memory your app can use is limited. On very old or very cheap devices, it might be as little as 16MB, though higher values (e.g., 32MB, 48MB) are a bit more common today. Still, there is always a cap. Having *one* screen use several MB of memory will be a problem for apps that have lots of screens, particularly if many of those screens will have their own long lists.

AdapterView and RecyclerView are designed around recycling. If you look at that screenshot again, no matter how many rows there might be in the list overall, the user can only see 8 of them on this particular screen at once. View recycling takes advantage of this, so we can limit our memory usage. Rather than having widgets for each item in the list, we have widgets for each *visible* item, plus a few more to help with reacting to scrolling events. As rows in the list scroll off the screen, they become eligible for reuse (recycling). This helps keep our memory usage to a more reasonable level, despite having a potentially very long list.

Enter RecyclerView

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:

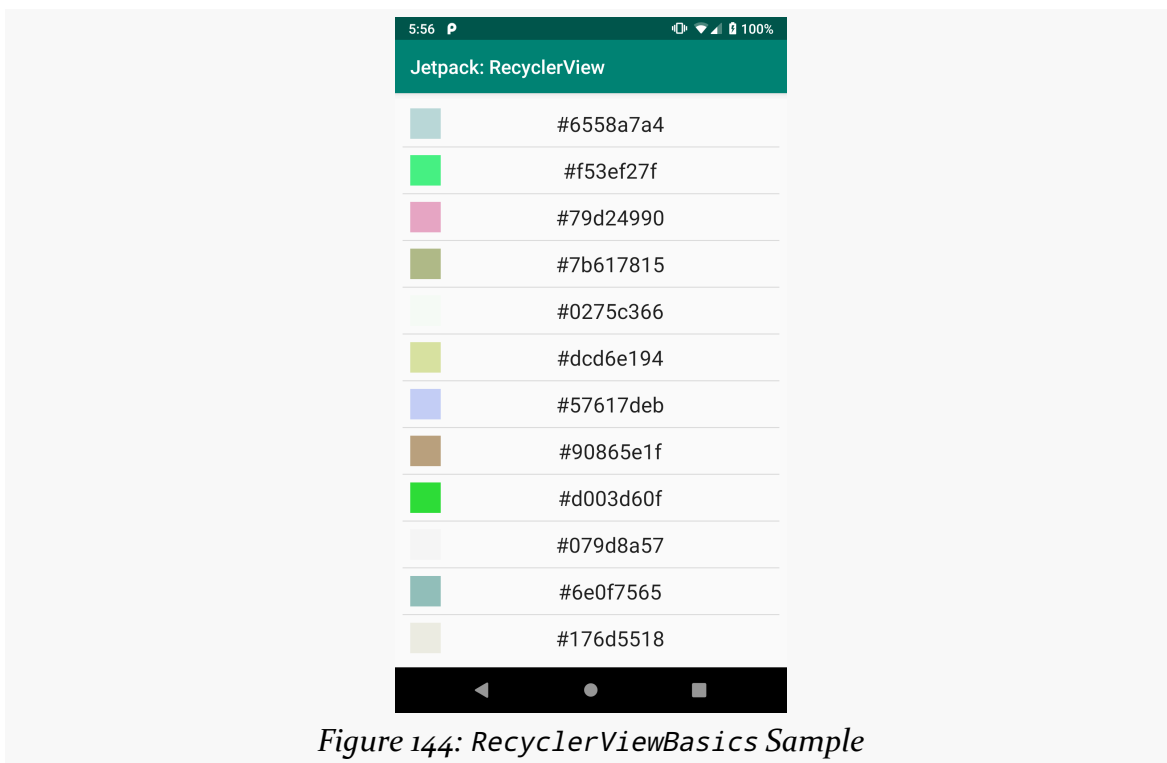
- a layout manager, responsible for organizing the views into various structures (vertical list, grid, staggered grid, etc.)
- an item decorator, responsible for applying effects and light positioning to the views, such as adding divider lines between rows in a vertical list
- an item animator, responsible for animated effects as the model data changes

- and so on

Through “adapter” and “view-holder” classes, we teach RecyclerView what should go into the list rows, grid cells, or whatever. RecyclerView then handles the scrolling and recycling for us.

A Trivial List

The RecyclerViewBasics sample module in the [Sampler](#) and [SamplerJ](#) projects demonstrate a fairly simple list. The items in the list will be a set of randomly-generated integers, where we will show the values both as a hexadecimal number and as a color:



The Dependency

RecyclerView comes from a library, not the framework portion of the Android SDK. As a result, we need a new dependency, one for `androidx.recyclerview:recyclerview:`

EMPLOYING RECYCLERVIEW

```
dependencies {  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.recyclerview:recyclerview:1.2.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.1'  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation "androidx.activity:activity:1.3.1"  
}
```

(from [RecyclerViewBasics/build.gradle](#))

Elsewhere, you may see references to `com.android.support:recyclerview-v7`. That library offers the same RecyclerView, but is part of the older Android Support Library. Jetpack projects should use the `androidx` edition of RecyclerView.

The Layouts

This project has two layout resources: `activity_main` and `row`.

The Activity Layout

The `activity_main` layout resource contains a RecyclerView inside of a ConstraintLayout:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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"  
    android:padding="@dimen/content_padding"  
    tools:context=".MainActivity">  
  
    <androidx.recyclerview.widget.RecyclerView  
        android:id="@+id/items"  
        android:layout_width="0dp"  
        android:layout_height="0dp"  
        app:layout_constraintBottom_toBottomOf="parent"  
        app:layout_constraintStart_toStartOf="parent"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintTop_toTopOf="parent" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [RecyclerViewBasics/src/main/res/layout/activity_main.xml](#))

EMPLOYING RECYCLERVIEW

As with `ConstraintLayout` itself, `RecyclerView` needs to have its fully-qualified class name in the layout resource XML, which is why we have an `<androidx.recyclerview.widget.RecyclerView>` element instead of simply a `<RecyclerView>` element. Otherwise, `RecyclerView` is just an ordinary thing that we can size and position as needed. Here, we have it set to fill the `ConstraintLayout`, excluding 8dp of padding that the `ConstraintLayout` applies.

Note that `RecyclerView` knows how to scroll, so we do not need to wrap `RecyclerView` in a `ScrollView`.

When constructing a new UI, if you wish to use the drag-and-drop GUI builder in Android Studio, you can find `RecyclerView` in the “Containers” category of the Palette:

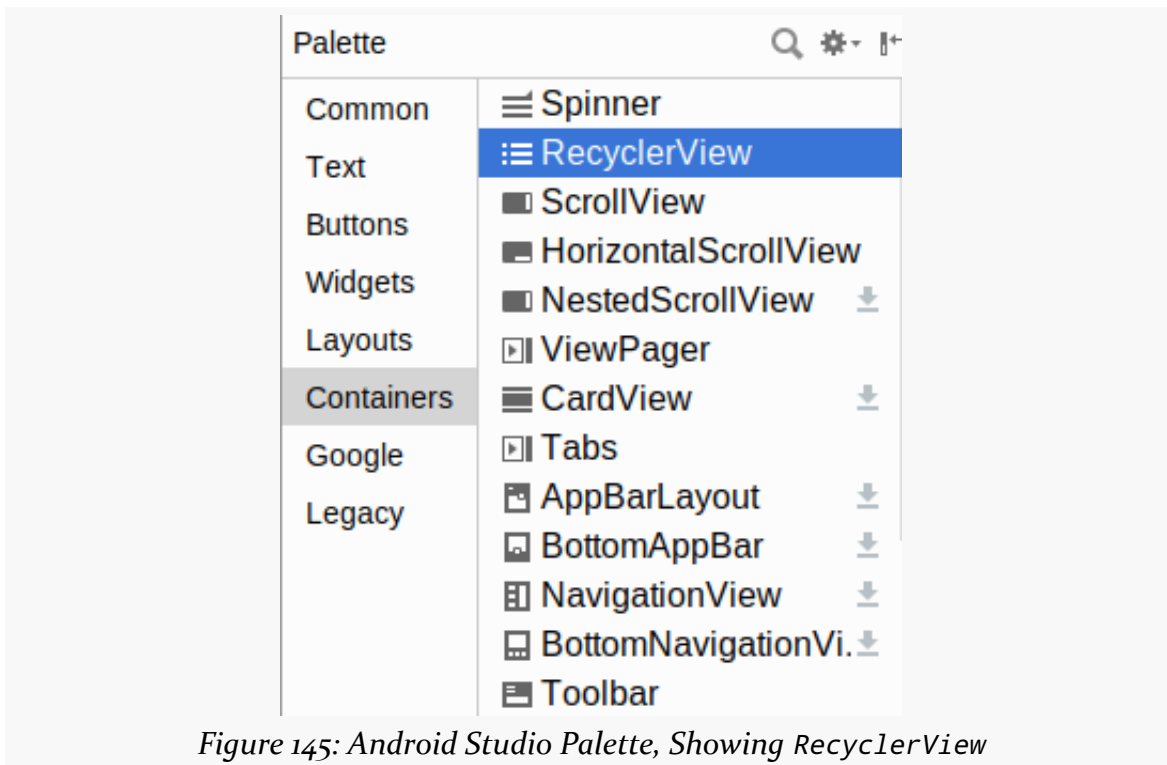


Figure 145: Android Studio Palette, Showing `RecyclerView`

The Row Layout

This project has a second layout resource, named `row`. Each row in our `RecyclerView` will be created from this row layout resource as a template:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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"
    android:background="?android:attr/selectableItemBackground"
    android:clickable="true"
    android:focusable="true"
    android:padding="@dimen/content_padding">

    <View
        android:id="@+id/swatch"
        android:layout_width="@dimen/swatch_size"
        android:layout_height="@dimen/swatch_size"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/label_start_margin"
        android:textAppearance="?android:attr/textAppearanceLarge"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@id/swatch"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [RecyclerViewBasics/src/main/res/layout/row.xml](#))

Once again, our root is a `ConstraintLayout`, this time with two children: a `TextView`... and a `View`.

Here, `View` is literally referring to `android.view.View`, the root class of all widgets and containers. `View` is not used all that often, as it does not render anything in the foreground: no text, no image, etc. However, it can have a background color, so the primary usage of `View` is for lines, boxes, and other shaded areas that have no children. In this case, we are using it for the color swatch seen on the left side of the rows, so we give it an ID of `swatch`.

The `label` `TextView` will be used to show the hexadecimal number that we randomly generate and are using for the color. Here, we use `android:textAppearance` as an attribute. That sets a bunch of things at once: text size, text style, etc. In particular,

our value (`?android:attr/textAppearanceLarge`) is a reference to a standard text appearance from our activity's theme. We will explore themes more [later in the book](#). For now, take it on faith that `?android:attr/textAppearanceLarge` will give us a standard “large” text appearance.

The `ConstraintLayout` has three attributes that we have not used before:

- `android:background="?android:attr/selectableItemBackground"`
- `android:clickable="true"`
- `android:focusable="true"`

The `android:clickable` and `android:focusable` attributes control whether the user can interact with this element. By default, container classes like `ConstraintLayout` are non-interactive, so they ignore any touch or key events. By setting `android:clickable="true"` and `android:focusable="true"`, we are indicating that this particular container is interactive, and so we should pay attention to touch and key events. This is typical of a container that serves as the root for an individual item in a `RecyclerView`.

In particular, those tie into `android:background="?android:attr/selectableItemBackground"`. `android:background` says “this is what the background of this thing should be”. The default background for a container like `ConstraintLayout` is transparent. In this case, we are replacing that with something that we are pulling from our theme (`?android:attr/selectableItemBackground`), which will apply a standard “selectable” background. If you run the sample apps, and you tap on a row in the list, you will see a ripple effect — this comes from the “selectable” background.

The LayoutManager

As with our other samples, our activity is `MainActivity`, whether that is written in Java:

```
package com.commonware.jetpack.samplerj.recyclerview;

import android.os.Bundle;
import com.commonware.jetpack.samplerj.recyclerview.databinding.ActivityMainBinding;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;

public class MainActivity extends AppCompatActivity {
```

EMPLOYING RECYCLERVIEW

```
private final Random random = new Random();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    final ActivityMainBinding binding =
        ActivityMainBinding.inflate(getLayoutInflater());

    setContentView(binding.getRoot());

    ColorAdapter adapter = new ColorAdapter(getLayoutInflater());

    adapter.submitList(buildItems());
    binding.items.setLayoutManager(new LinearLayoutManager(this));
    binding.items.addItemDecoration(
        new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
    binding.items.setAdapter(adapter);
}

private List<Integer> buildItems() {
    ArrayList<Integer> result = new ArrayList<>(25);

    for (int i = 0; i < 25; i++) {
        result.add(random.nextInt());
    }

    return result;
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/MainActivity.java](#))

... or Kotlin:

```
package com.commonsware.jetpack.sampler.recyclerview

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.jetpack.sampler.recyclerview.databinding.ActivityMainBinding
import java.util.*

class MainActivity : AppCompatActivity() {
    private val random = Random()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)

        binding.items.apply {
            layoutManager = LinearLayoutManager(this@MainActivity)
            addItemDecoration(
                DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL)
            )
        }
    }
}
```

EMPLOYING RECYCLERVIEW

```
        adapter = ColorAdapter(layoutInflater).apply {  
            submitList(buildItems())  
        }  
    }  
}  
  
private fun buildItems() = List(25) { random.nextInt() }  
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/MainActivity.kt](https://github.com/commonsware/jetpack-sampler/recyclerview/blob/master/src/main/java/com/commonsware/jetpack/sampler/recyclerview/MainActivity.kt))

The UI is set up in `onCreate()`. Here, we configure the `RecyclerView`.

One thing that we need to teach the `RecyclerView` is what sort of look we want it to have overall. In this case, we want a vertically-scrolling list. For that, we can use a `LinearLayoutManager`.

`RecyclerView` uses a `LayoutManager` to control the overall layout of items within its scrollable area. `LinearLayoutManager` lays those items out in a list. By default, that list is vertical.

(despite the name, `LinearLayoutManager` is not related to `LinearLayout`, except in the most general sense)

So, we create an instance of `LinearLayoutManager` and associate it with the `RecyclerView` via `setLayoutManager()`.

If you do not want a vertically-scrolling list, you could use:

- `GridLayoutManager`, which implements a two-dimensional vertically-scrolling list, with rows consisting of multiple cells
- `StaggeredGridLayoutManager`, which implements a “staggered grid”, which has columns of cells, but where the cells do not have to all have the same size

In addition, it is possible to create your own `RecyclerView.LayoutManager`, or use ones from third-party libraries.

The Divider

By default, a `RecyclerView` just shows its items, and nothing else. However, with a vertically-scrolling list, frequently we want some sort of divider between the rows in the list. Sometimes, that is not necessary, as we have stuff in the items themselves that visually distinguish one from another. The rest of the time, we will want the

help of RecyclerView to handle this.

RecyclerView delegates that sort of work to one or more `ItemDecoration` objects. The primary one offered by the RecyclerView library is `DividerItemDecoration`, which draws a thin line for you.

`DividerItemDecoration` can be used both for the normal vertically-scrolling list and for the less-common horizontally-scrolling list. Hence, we need to tell `DividerItemDecoration` which way to draw the divider line. For a vertically-scrolling list, we need a divider configured for such a list, so we pass `DividerItemDecoration.VERTICAL` to the `DividerItemDecoration` constructor when we set it up.

You can then add these `ItemDecoration` objects to a `RecyclerView` via `addItemDecoration()` calls.

The Data

The data that we want to depict in the list is a roster of 25 random numbers. To that end, we have a `buildItems()` Java method:

```
private List<Integer> buildItems() {
    ArrayList<Integer> result = new ArrayList<>(25);

    for (int i = 0; i < 25; i++) {
        result.add(random.nextInt());
    }

    return result;
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/MainActivity.java](#))

... or Kotlin function:

```
private fun buildItems() = List(25) { random.nextInt() }
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/MainActivity.kt](#))

In each case, we create a list of 25 random numbers, using `Random.nextInt()` as the source of the number. `buildItems()` is called from `onCreate()` and will be used to populate our `RecyclerView.Adapter`.

The ViewHolder

Each RecyclerView is associated with one or more implementations of RecyclerView.ViewHolder. Each instance of a ViewHolder class wraps one piece of data (in our case, one random number) and ties it to one visual representation of that data. However, courtesy of recycling, a ViewHolder will get reused. As the user scrolls, and items scroll off the screen, their associated ViewHolder objects will need to be associated with new pieces of data that are to be scrolled onto the screen.

Our particular implementation is ColorViewHolder, both in Java:

```
package com.commonware.jetpack.samplerj.recyclerview;

import android.widget.Toast;
import com.commonware.jetpack.samplerj.recyclerview.databinding.RowBinding;
import androidx.recyclerview.widget.RecyclerView;

class ColorViewHolder extends RecyclerView.ViewHolder {
    private final RowBinding binding;

    ColorViewHolder(RowBinding binding) {
        super(binding.getRoot());

        this.binding = binding;

        binding.getRoot().setOnClickListener(
            v -> Toast.makeText(binding.label.getContext(), binding.label.getText(),
                Toast.LENGTH_LONG).show());
    }

    void bindTo(Integer color) {
        binding.label.setText(
            binding.label.getContext().getString(R.string.label_template, color));
        binding.swatch.setBackgroundColor(color);
    }
}
```

(from [RecyclerViewBasics/src/main/java/com/commonware/jetpack/samplerj/recyclerview/ColorViewHolder.java](#))

... and Kotlin:

```
package com.commonware.jetpack.sampler.recyclerview

import android.widget.Toast
import androidx.recyclerview.widget.RecyclerView
import com.commonware.jetpack.sampler.recyclerview.databinding.RowBinding
```

```
class ColorViewHolder(private val binding: RowBinding) :  
    RecyclerView.ViewHolder(binding.root) {  
    init {  
        binding.root.setOnClickListener { _ ->  
            Toast.makeText(  
                binding.label.context,  
                binding.label.text,  
                Toast.LENGTH_LONG  
            ).show()  
        }  
    }  
}  
  
fun bindTo(color: Int) {  
    binding.label.text =  
        binding.label.context.getString(R.string.label_template, color)  
    binding.swatch.setBackgroundColor(color)  
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/ColorViewHolder.kt](#))

The Constructor

Something outside of the ViewHolder is responsible for setting up the visual representation and providing the View for it to our constructor. We will see how that works [a bit later in this chapter](#). Our ColorViewHolder is expecting this View to be the root of our row layout resource, so we can work with its contents.

The job of our constructor is to find the individual widgets that we need and configure them for our use.

Here, we pass in our RowBinding and get the root from it to pass to the ViewHolder superclass.

The Toast

We also call `setOnClickListener()` on the row itself, to find out when it is clicked. There, to provide feedback to the user, we show a Toast.

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, ones that if the user misses them, no harm will come.

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.

The Binding

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. However, at some point, our `ViewHolder` needs to be handed the data that it is supposed to represent in these widgets.

For that, `ColorViewHolder` has `bindTo()`. It takes a color integer and pours it into the label and swatch widgets.

For the label, we use `getString()`, where our string resource has a placeholder to format the number:

```
<string name="label_template">#%1$08x</string>
```

(from [RecyclerViewBasics/src/main/res/values/strings.xml](#))

That cryptic placeholder (`#%1$08x`) means that we want to format the first parameter (`%1`) as an eight-digit hexadecimal value (`08x`).

For the color swatch, we call `setBackgroundColor()`. This is a method available on any `View` that can be used to set its background color to a particular value. In our case, we are using a randomly-generated color, not a color resource, so we can just pass in the color integer.

The Adapter

We still need to tell the `RecyclerView` what to display. That is handled by an implementation of `RecyclerView.Adapter`. Here, “adapter” refers to the adapter

pattern: an adapter takes data and adapts it for some other role. In our case, an Adapter takes some collection of data and uses it to define the visual representation of that data, in the form of View and ViewGroup objects.

Our Adapter — named ColorAdapter — will use the row layout that we defined earlier to define the visual representation of each piece of data. ColorAdapter uses our ColorViewHolder to manage the actual widgets.

We have two implementations of ColorAdapter, in Java:

```
package com.commonware.jetpack.samplerj.recyclerview;

import android.view.LayoutInflater;
import android.view.ViewGroup;
import com.commonware.jetpack.samplerj.recyclerview.databinding.RowBinding;
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.DiffUtil;
import androidx.recyclerview.widget.ListAdapter;

class ColorAdapter extends ListAdapter<Integer, ColorViewHolder> {
    private final LayoutInflater inflater;

    ColorAdapter(LayoutInflater inflater) {
        super(DIFF_CALLBACK);
        this.inflater = inflater;
    }

    @NonNull
    @Override
    public ColorViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                             int viewType) {
        return new ColorViewHolder(RowBinding.inflate(inflater, parent, false));
    }

    @Override
    public void onBindViewHolder(@NonNull ColorViewHolder holder, int position) {
        holder.bindTo(getItem(position));
    }

    private static final DiffUtil.ItemCallback<Integer> DIFF_CALLBACK =
        new DiffUtil.ItemCallback<Integer>() {
            @Override
            public boolean areItemsTheSame(@NonNull Integer oldColor,
                                           @NonNull Integer newColor) {
                return oldColor.equals(newColor);
            }
        }
}
```



```
@Override
public boolean areContentsTheSame(@NonNull Integer oldColor,
                                   @NonNull Integer newColor) {
    return areItemsTheSame(oldColor, newColor);
}
};
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/ColorAdapter.java](#))

... and Kotlin:

```
package com.commonsware.jetpack.sampler.recyclerview

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import com.commonsware.jetpack.sampler.recyclerview.databinding.RowBinding

class ColorAdapter(private val inflater: LayoutInflater) :
    ListAdapter<Int, ColorViewHolder>(ColorDiffer) {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): ColorViewHolder {
        return ColorViewHolder(RowBinding.inflate(inflater, parent, false))
    }

    override fun onBindViewHolder(holder: ColorViewHolder, position: Int) {
        holder.bindTo(getItem(position))
    }

    private object ColorDiffer : DiffUtil.ItemCallback<Int>() {
        override fun areItemsTheSame(oldColor: Int, newColor: Int): Boolean {
            return oldColor == newColor
        }

        override fun areContentsTheSame(oldColor: Int, newColor: Int): Boolean {
            return areItemsTheSame(oldColor, newColor)
        }
    }
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/ColorAdapter.kt](#))

The Base Class

When we created `ColorViewHolder`, we directly extended `RecyclerView.ViewHolder`. You can do the same thing with `RecyclerView.Adapter`, having your class (e.g., `ColorAdapter`) extend it directly. In this case, we are using `ListAdapter`, a supplied partial implementation of `RecyclerView.Adapter` that knows how to work with lists of data.

NOTE: There are *two* things named `ListAdapter` in the Android SDK. We are using `androidx.recyclerview.widget.ListAdapter`, which works with `RecyclerView`. The `android.widget.ListAdapter` interface is designed for use with [the older `AdapterView` family of widgets](#), such as `ListView`. Make sure that you use the right one, as otherwise you will get lots of strange compile errors.

`ListAdapter` uses generics and takes two data types:

- The type of data for individual elements in the list to be shown (in our case, a Java `Integer` or Kotlin `Int`)
- The `RecyclerView.ViewHolder` class that we want to use for the items in the list (in our case, `ColorViewHolder`)

The Constructor and the “Differ”

The `ListAdapter` constructor requires an implementation of a `DiffUtil.ItemCallback` object.

Part of what `ListAdapter` does for us is help deal with changes to our list of data. This app, as it stands, only shows one set of random numbers in the list. But, suppose we had a button that allowed the user to add more numbers. We would need our `RecyclerView.Adapter` to be able to show both the old numbers and the new numbers.

`ListAdapter` has all of the smarts to handle that for us as efficiently as possible. However, `ListAdapter` knows nothing about our data and our visual representation of that data, and knowing more about those things helps with efficiency. The `DiffUtil.ItemCallback` is our way of teaching `ListAdapter` more about our data. Specifically, a `DiffUtil.ItemCallback` has two Java methods or Kotlin functions:

- `areItemsTheSame()` takes in two pieces of data from our list (old and changed) and needs to return `true` if they are the actual same thing
- `areContentsTheSame()` takes in two pieces of data and returns `true` if their

visual representation will be the same when the data is rendered on the screen

If our `ListAdapter` were adapting the items in an online shopping cart, we would return `true` from `areItemsTheSame()` if both objects are really the same underlying cart entry. But, suppose the user had more than one of some particular product in the cart, such as three boxes of laundry detergent. `areItemsTheSame()` might return `false` for some pair, as the first box is not the same box as the second box. However, `areContentsTheSame()` might return `true`, as the visual representation might be the same (e.g., a thumbnail image of the laundry detergent box).

In our case, not only is the data not changing, but it is very simple and distinct, so we can use content equality for both `areItemsTheSame()` and `areContentsTheSame()`. So, we have a `ColorDiffer` that does just that, either in the form of a static Java class or a singleton Kotlin object. We use `ColorDiffer` in our `ColorAdapter` constructor, so now `ListAdapter` knows about how to compare our colors.

`onCreateViewHolder()`

Any `RecyclerView.Adapter` needs to know how to do two other things:

1. Create the `ViewHolder` that we want, including setting up its UI; and
2. Bind data from our collection to `ViewHolder` instances as they are displayed on the screen

`onCreateViewHolder()` handles the first of these. It creates instances of `ColorViewHolder` and returns them.

However, `ColorViewHolder` wants the `View` that represents the UI for the row in our list. That UI is defined in the row layout resource. We need some way to get a `View` for a layout resource. All our prior uses of layout resources were for activities, and we just passed the resource ID to `setContentView()`. That will not work here.

So, our `ColorAdapter` constructor takes in a `LayoutInflater` object. `LayoutInflater` knows how to “inflate” layout resources. In Android, “inflate” means:

- Walk a tree of XML elements in an XML resource
- Create Java objects for each element in that tree
- Stitch those objects together into their own tree structure, mirroring the tree defined in the XML

- Return the Java object representing the root of the tree

Given that our row layout is:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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"
    android:background="?android:attr/selectableItemBackground"
    android:clickable="true"
    android:focusable="true"
    android:padding="@dimen/content_padding">

    <View
        android:id="@+id/swatch"
        android:layout_width="@dimen/swatch_size"
        android:layout_height="@dimen/swatch_size"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/label_start_margin"
        android:textAppearance="?android:attr/textAppearanceLarge"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@id/swatch"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [RecyclerViewBasics/src/main/res/layout/row.xml](#))

...then a call to `inflate()` on a `LayoutInflater`, supplying `R.layout.row` as the first parameter, should return a `ConstraintLayout` object that holds onto a `TextView` and a `View`.

That is what `onCreateViewHolder()` does: calls `inflate()`, gets the root `View`, passes that to the `ColorViewHolder()` constructor, and returns the `ColorViewHolder` instance.

`LayoutInflater` has a few `inflate()` methods, taking different parameters. They all

do the same basic thing: “inflate” the layout into a View hierarchy. The particular `inflate()` that you will use most often takes three parameters:

- The resource ID of the layout that you want to inflate
- The ViewGroup that the widgets in that layout will eventually be added to
- `false` to say “but please do not add them to the parent right away” (something else will do that when appropriate, RecyclerView and LinearLayoutManager in this case)

In our case, we are using view binding, which means that our binding object (e.g., `RowBinding`) is the one using the `LayoutInflater`, and we call a similar `inflate()` method on the binding object.

`onBindViewHolder()`

`onBindViewHolder()` will be called when RecyclerView wants to show one of our pieces of data. We are passed the 0-based index into our collection of data representing what RecyclerView wants, and we are passed a `ViewHolder` (created by `onCreateViewHolder()` previously) to use for the visual representation.

`ListAdapter` gives us a `getItem()` method that we can use to get our color for a given position. We then just call `bindTo()` on the `ColorViewHolder`, and `ColorViewHolder` takes it from there.

Applying the ColorAdapter

To make use of `ColorAdapter`, we first need to give it some colors to display. Subclasses of `ListAdapter` have a `submitList()` method that you can use for that. You provide a `List` of your data, such as the results of our `buildItems()` call.

Then, to have the colors show up, you call `setAdapter()` on the RecyclerView, handing it your `RecyclerView.Adapter` instance.

We do both of these things in `onCreate()` of `MainActivity` as part of the overall setup:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    final ActivityMainBinding binding =
```

```
ActivityMainBinding.inflate(getLayoutInflater());

setContentView(binding.getRoot());

ColorAdapter adapter = new ColorAdapter(getLayoutInflater());

adapter.submitList(buildItems());
binding.items.setLayoutManager(new LinearLayoutManager(this));
binding.items.addItemDecoration(
    new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
binding.items.setAdapter(adapter);
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/MainActivity.java](#))

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityMainBinding.inflate(layoutInflater)

    setContentView(binding.root)

    binding.items.apply {
        layoutManager = LinearLayoutManager(this@MainActivity)
        addItemDecoration(
            DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL)
        )
        adapter = ColorAdapter(layoutInflater).apply {
            submitList(buildItems())
        }
    }
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/MainActivity.kt](#))

Hey, What About ListView?

If you read about Android from older sources, you will see a lot of discussion of `ListView`.

`ListView` is a subclass of `AdapterView`. The `AdapterView` family of classes is the precursor to `RecyclerView`. Like `RecyclerView`, they display collections of data. And, like `RecyclerView`, they use an adapter to convert your data into visual representations to display. However, on the whole, the `AdapterView` widgets were less configurable and less powerful than is `RecyclerView`.

Almost every `AdapterView` subclass that is part of the Android SDK could be replaced by `RecyclerView`. `ListView` and `GridView` can be replaced very easily. `Gallery` would take a bit more work. Many third-party libraries exist for

implementing tree structures — parents with children, like a directory tree — in RecyclerView, to replace ExpandableListView.

The main exception is Spinner, which implements a drop-down list. RecyclerView cannot reproduce this sort of widget directly, though it could be used as part of some larger replacement.

Gesture Navigation and Scrolling Widgets

Some Android users — including many who purchase an Android 10+ device — will wind up using a “gesture” form of system navigation. This can interfere with scrolling widgets, like a RecyclerView.

A Tale of Three (or More) Nav Patterns

Way back in the beginning, navigation actions were handled by hardware buttons. Android 3.0 introduced the notion of a “navigation bar” for handling “home”, “back”, and “overview” navigation actions, leading to the classic three-button bar:

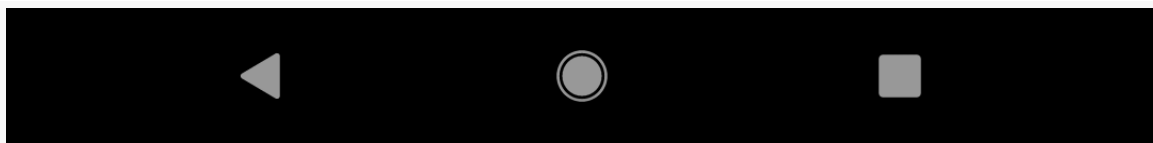


Figure 146: Three-Button Android Nav Bar

Android 9.0 added another option for users: a two-button nav, where “home” and “overview” actions were handled by gestures on a central pill affordance:

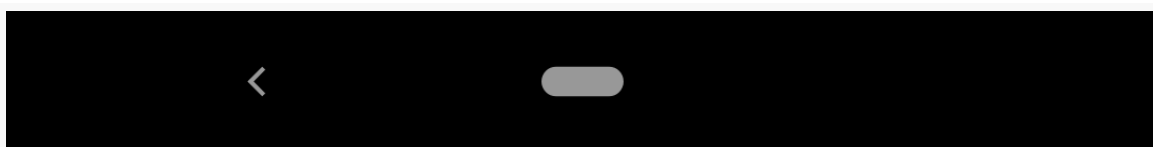


Figure 147: Button-and-Pill Android 9.0 Nav Bar

Android 10 deprecates that two-button nav option but adds a new nav option that is based on gestures:

EMPLOYING RECYCLERVIEW

Action	Associated Gesture
Home	swipe up from bottom screen edge
Back	swipe inward from the screen edge on left or right
Overview	swipe up from the bottom screen edge and hold

Users can choose among those by visiting Settings > System > Gestures > “System navigation”:

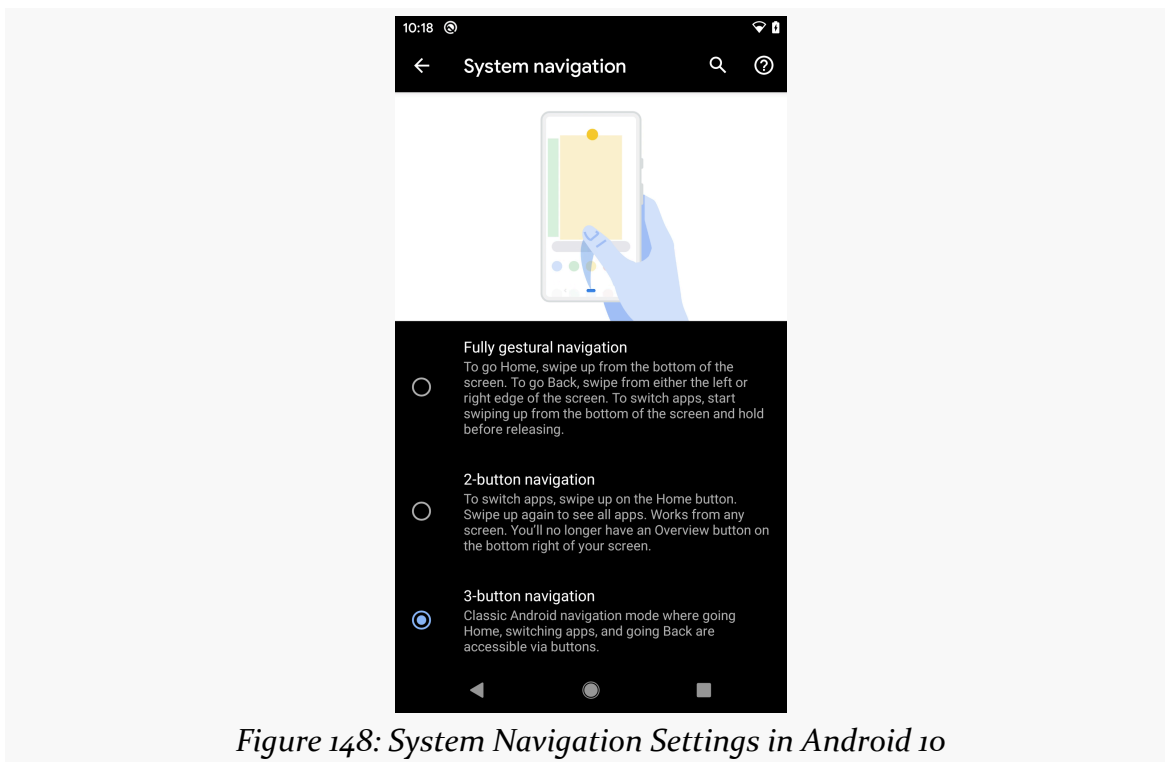


Figure 148: System Navigation Settings in Android 10

The user can choose between gesture-based nav, the Android 9.0 button-and-pill option, or the classic three-button nav option. Note, though, that not all users will have access to all of those options. Pixel 4 users, for example, cannot choose the two-button nav option.

On top of this, some device manufacturers have created their own gesture-based nav options. Device manufacturers will be allowed to continue coming up with their own schemes for this, meaning that a user might have three or four navigation options

on Android 10 devices.

Impacts on Apps

The system “steals” touch events from apps to handle these navigation gestures. If your app relies upon touch events near the edges, you may run into some problems. In particular, the user may get confused when trying to use your app, trying to apply *your* gestures and winding up with system responses. While simple taps will be passed through to your app from these system edge areas, anything else is indeterminate.

For example, suppose that you have a RecyclerView that goes all the way to the bottom of the screen. Based on a subtle and invisible line of demarcation, the same gesture might either scroll your RecyclerView or take the user to the home screen. Those are substantially different results for the same gesture, applied in slightly different locations.

You may need to consider redesigning your UI to:

- Avoid expecting swipe gestures near screen edges, and
- Provide a visual distinction of where swipe gestures are valid, to help the user learn where to swipe to control your UI

Technically, [there is a way](#) that you can tell the system to ignore “back” gestures and pass those along to your app. However, from a practical standpoint, this has problems:

- The user may not know how to exit this screen and may get frustrated as a result
- This approach may not be honored by manufacturer-specific nav schemes

Avoiding the edges is a safer approach.

The OS informs your app about “window insets”, to indicate areas where the system will steal your touch events. [This library](#) helps you leverage that information to adjust your UI based upon the particular device’s window insets, based on device model and whether the user has enabled gesture-based nav or not.

On the whole, this book’s samples ignore the impacts of gesture navigation.

Coping with 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 switch to a different language via the Settings application, returning to our running application afterwards
- It might become dark, suggesting that we should be in some sort of “night mode”
- 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. The core Android team does not expect that you will want different resources based on whether or not the device has a front-facing camera.

Most of the hardware features that drive a configuration might change on the fly. For example, the user can rotate the device and switch from portrait to landscape, while our app is still running. Somehow, though, we may need to switch to landscape-friendly resources from the portrait-friendly resources that we started with. When a configuration switches to something else on the fly, 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

Somehow, we need to be able to tell Android:

- Some resources are to be used for a certain type of configuration, such as "large screens" or "high-density screens" or "uses English"
- Some resources are for some other configuration, for as many different configuration aspects as we wish to support
- Some resources are the default and are valid for any configuration

The way Android currently handles this is by having multiple resource directories, with the criteria for each embedded in their names.

For example, suppose that you want to support multiple languages. You will need to choose some default language, one that will be used if your app winds up on a device whose locale is set to a language that you do not support. For example, you might set your default language to be US English. In that case, your US English

strings would go into `res/values/strings.xml`. If you also wanted to offer a translation in Spanish, you would add a `res/values-es/` directory, where `es` is the [ISO 639-1](#) two-letter code for Spanish. In `res/values-es/`, you would put your Spanish strings. At runtime, Android will choose which string to use, for a given string resource ID (e.g., `@string/app_name`), based on the device's locale(s) and the various translations of that string that you provide.

This, therefore, is the bedrock resource set strategy: have a complete set of resources in the default directory (e.g., `res/values/`), and override those resources in other resource sets tied to specific configurations as needed (e.g., `res/values-es/`).

Implementing Resource Sets

Ideally, that strategy would be sufficient. Unfortunately, things are not quite that simple. So, let's walk through some common configurations and associated resource sets, to see how we can set them up.

Language

As noted above, you can have different resource sets for different languages, by adding the language code as a suffix on the resource directory name. Usually, this is just applied to `res/values/` (e.g., `res/values-es/`), as usually the only things that change in an app based on language are string resources. In principle, though, you could use such language suffixes on any resource directory (e.g., `res/raw/` for English-language audio clips and `res/raw-es/` for Spanish-language audio clips).

However, languages often vary by region:

- US English has much in common with UK English, but not everything (e.g., “color” versus “colour”)
- French as spoken in France differs somewhat from French as spoken in the Quebec province of Canada
- Spanish as spoken in Spain differs somewhat from Spanish as spoken in Mexico
- And so on

To address this, Android supports two different systems for regional localization.

The classic approach was to add an [ISO-3166-1 alpha-2](#) code after the language code and prefixed by `r` (for “region”). For example, US English is `res/values-en-rUS/`,

while UK English is `res/values-en-rGB/` (“GB” for “Great Britain”).

Android 7.0 and higher support [BCP 47 language and locale values](#) as an alternative system. Those get an overall `b+` prefix, to distinguish them from other resource set qualifiers. In the BCP 47 approach, English is `res/values-b+en/`, while US English is `res/values-b+en+US/` and UK English is `res/values-b+en+GB/`. However, this system has not been as popular, in part because we have so much history with the original system.

In either case, string resource sets are additive. You do not need to have a full set of US English strings *and* a full set of UK English strings to support full localization, because many of those strings will be in common. Plus, you can choose one of the regions to be the default for a particular language, so you only need resource sets and overrides for the specific strings that you need for the specific locales that concern you.

So, for example, you could have:

- US English strings in `res/values/` as the overall default for your app
- UK English strings in `res/values-en-rGB/`, for the handful of strings that you need where the British spelling or term differs from the US version
- Mexican Spanish strings in `res/values-es/`
- Spain Spanish strings in `res/values-es-rES/`, for the handful of strings that you need where the Spain localization differs from the Mexican localization
- And so on

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

The resource sets that get the most attention are those for screen size (and, secondarily, screen orientation).

Android “borrows a page” from Web development. Web content nowadays uses CSS media queries to have different layout rules based on screen size, typically screen width. Similarly, Android allows you to set up different resource sets to use for screens with varying sizes, where you choose the size. And, like CSS media queries, you can choose the dividing lines between different rules — you are not limited to some fixed set of size buckets.

There are three families of qualifiers that you can use:

- `-wNNNdp`, for some number `NNN`, means “use these resources when the current width of the screen is `NNN` density-independent pixels (dp) or larger”
- `-hNNNdp`, for some number `NNN`, means “use these resources when the current height of the screen is `NNN` dp or larger”
- `-swNNNdp` means “use these resources when the *smallest width* of the screen, in any orientation, is `NNN` dp or larger”

The difference between `-wNNNdp` and `-swNNNdp` is that `-wNNNdp` is based on the *current* width, and the current width of a device depends on its orientation. `-swNNNdp`, by contrast, cares only what the length of the shorter screen dimension is (“smallest width”) and therefore is independent of orientation.

So, you could have:

- `res/layout/` for layout resources aimed at smaller screens
- `res/layout-w640dp/` for layout resources aimed at screens whose current width is 4" (640dp) or larger
- `res/layout-w1120dp/` for layout resources aimed at screens whose current width is 7" (1120dp) or larger

Hey, What About `res/layout-large`?

You will see references to other size-related resource set qualifiers: `-small`, `-normal`, `-large`, and `-xlarge`. This was the original system used by Android. However, it was not very flexible. Android 3.2 added the system described above, and so most modern Android app development should use that system.

Hey, What About `res/layout-land`?

Similarly, you will see references to a `-land` suffix to be used for landscape resources. So `res/layout/` would be used for portrait layouts and `res/layout-land/` would be used for landscape layouts.

Frequently, though, we do not care about the actual orientation, only the width (or *maybe* the height). For example, many Web sites that use CSS media queries apply different CSS rules based on the viewport width. While most computer monitors are landscape, some can be rotated to portrait — the CSS media queries do not care and only differentiate based on width.

As a result, while `-land` exists and can be used, you will not see it quite as much as you might have in the early days of Android app development.

API Level

Earlier in the book, we saw `res/drawable-v24/` as a resource directory. This follows the same resource set system, where `-vNN`, for some value of `NN`, means “use these resources on devices running API Level `NN` or higher”. So, `res/drawable-v24/` will be used on API Level 24+ devices and will be ignored on older devices.

Screen Density

Similarly, we saw various mipmap directories, like `res/mipmap-hdpi/` and `res/mipmap-xhdpi/`. This too follows the same resource set system, where `-hdpi` says “these resources were optimized for HDPI screens”, while `-xhdpi` says “these resources were optimized for XHDPI screens”.

The Full Roster

There are *lots* of different resource set qualifiers.

The full roster can be found in [the Android developer documentation](#). In particular, “Table 2” in that section lists all of the current candidates.

Note that resource set qualifiers get added from time to time. For example, the `-wNnndp` family of screen size qualifiers was added in API Level 13 (Android 3.2). The table points out what version of Android added support for those qualifiers. Older devices will not crash if they encounter those qualifiers, simply ignoring them instead.

Resource Set Rules

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, because they tie into local iconography (e.g., roadside traffic signs). 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 an American set of icons)

Once you get into these sorts of situations, though, a few rules come into play, such as:

- 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 density is more important than language.
- There can only be one value of each configuration option category per directory.

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` layout resource in:

- `res/layout-w640dp/`
- `res/layout/`

When we call `setContentView(R.layout.main)`, Android will choose the `main.xml` in `res/layout-w640dp/` if the device's current width is 640dp or larger. That particular resource set is valid in that case, and it has the most important distinct qualifier (`-w640dp`). If the device has a smaller current width, though, the `res/layout-w640dp/` resource set does not qualify, and so it is ignored. 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-w640dp/`
- `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 “Available width” (for `-w640dp`). 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-w640dp-v21/`
- `res/layout/`

You might think that `res/layout-w640dp-v21/` would be a higher-priority choice than `res/layout-en/`, 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-w640dp-v21/` for 4"-wide API Level 21+ 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-w640dp-night/`
- `res/layout-w640dp-v21/`
- `res/layout/`

Here, while `-w640dp` is the most important resource set qualifier, it is not distinct — we have more than one resource set with `-w640dp`. 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-w640dp-v21/` if the device is running API Level 21 or higher. If the device is not in night mode and is not running API Level 21 or higher — or if the device is less than 4" wide at present — Android will go with `res/layout/`.

Scenario #5: Screen Density

Now, let's look at the main 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 and mipmap 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 and mipmaps, 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-nonnav/`
- `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 drawable or mipmap resources, Android would upsample the `-mdpi` image.

Activity Lifecycles

The user taps a launcher icon to start our activity. Then, the user rotates the screen, causing a configuration change. Later, the user presses BACK to return to the launcher.

While those things were going on, Android was calling lifecycle methods on our activity, to let us know what is going on.

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

COPING WITH CONFIGURATIONS

another activity is overlaying part of the screen. During this time, the user can see parts of your activity but may not be able to interact with it.

3. *Stopped*: the activity was started by the user, is running, but it is completely hidden by other activities that have been launched or switched to.
4. *Destroyed*: the activity was destroyed, perhaps due to the user pressing the BACK button.

Android will call so-called “lifecycle methods” on your activity as the activity transitions between these four states.

[The Activity developer documentation](#) usually provides some variation of this diagram:

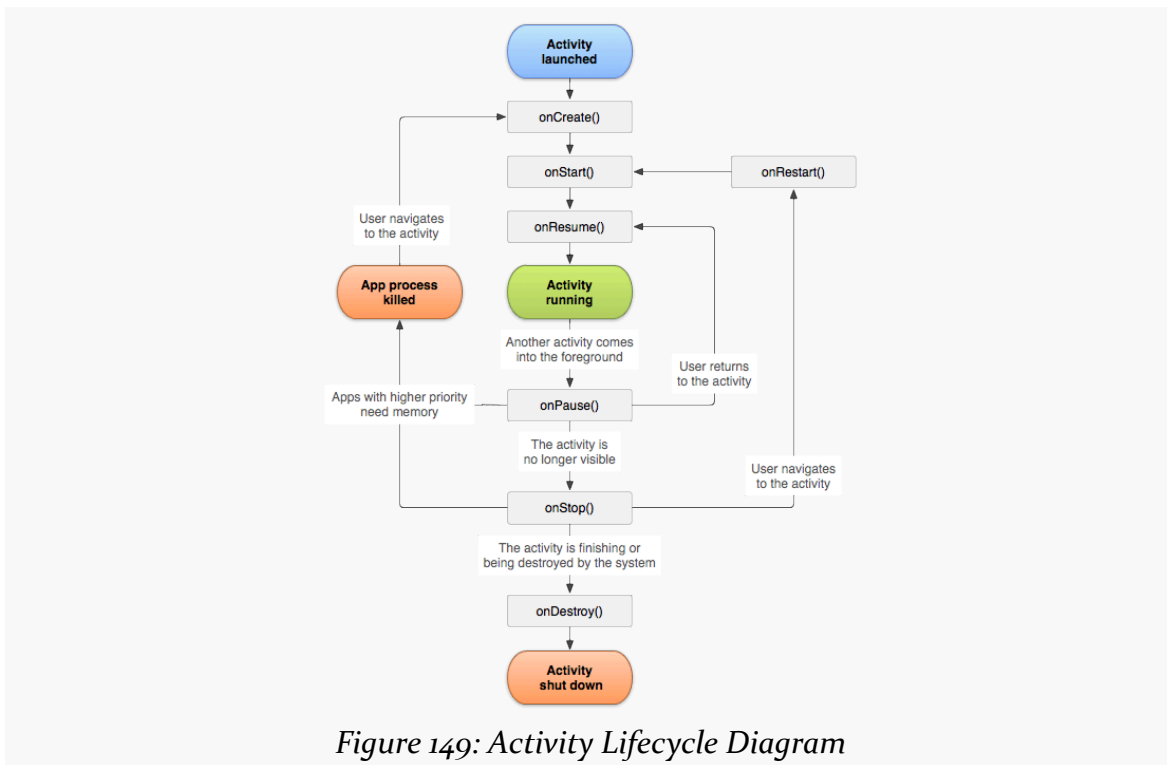


Figure 149: Activity Lifecycle Diagram

(the above image is 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](#))

This diagram shows the various lifecycle methods and the trigger events that cause them.

The following sections outline these lifecycle methods and their roles. We will see them in use starting in [the next chapter](#).

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 launched by the user, such as from a launcher icon, `onCreate()` will be invoked with a `null` parameter
- If the activity undergoes a configuration change, by default your activity will be re-created and `onCreate()` will be called

In general, `onCreate()` 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 app's activity).

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. Primarily, in `onStop()`, you clean up anything you set up in `onStart()`.

Once started, your activity is visible, at least partially. Anything that should be happening while your activity is visible should be set up in `onCreate()` or `onStart()` and cleaned up in `onStop()` (for `onStart()`) or `onDestroy()` (for `onCreate()`).

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. When your activity is resumed and is now fully in the foreground, the user can interact with it:

- They can tap on your widgets
- Navigation button clicks, such as BACK, affect your activity
- Hardware input, such as from a keyboard, is sent to your activity

Conversely, anything that takes over user input — the activation of another activity — will result in your `onPause()` being called. Here, you should undo anything you did in `onResume()`.

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.

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 `onStart()/onStop()`). Just do not mix and match between them.

Making the Superclass Happy

If you override a lifecycle method, you need to chain to the superclass' implementation of the method. Otherwise, you will crash at runtime with a `SuperNotCalledException`. Android Studio will warn you if you implement a lifecycle method and fail to chain to the superclass.

In practice, *when* you chain to the superclass' implementation is up to you, so long as it is in the same method (e.g., chaining to `super.onCreate()` from `onCreate()`). In theory, though, if you are relying on things that you inherit from `Activity`, it is safest to:

- Chain to the superclass before doing your own work for the creation set of methods (`onCreate()`, `onStart()`, `onResume()`)
- Chain to the superclass after doing your own work for the destruction set of methods (`onPause()`, `onStop()`, `onDestroy()`)

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 the next 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.

Context **Anti-Pattern: Outliving It**

Suppose that we had an activity that looked like this:

```
private lateinit var doNotDoThis: View

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)

        doNotDoThis = showElapsed // from the SimpleBoom layout, showElapsed is a widget
    }
}
```

We take a widget from our activity’s layout and we assign it to a global property.

On the surface, this may not seem all that bad. However, when the activity is destroyed — due to a configuration change, BACK button press, etc. — we now have a memory leak.

Each widget holds a reference to the activity that created it. In this case, `doNotDoThis` has a reference back to our `MainActivity`. Since `doNotDoThis` is global

in scope, that widget cannot be garbage collected. And since the `doNotDoThis` reference prevents the widget from being garbage collected, it prevents the destroyed activity from being garbage collected.

Fortunately, Android Studio will complain if you try to do this sort of thing.

In general, be *very* careful about having objects that are tied to some Context outlive the Context itself. This can include forking a background thread that has a reference to a Context, as that Context cannot be garbage collected until the thread terminates.

Integrating ViewModel

If you happened to install and run [the RecyclerViewBasics sample app](#), try the following:

1. Run the app
2. Make note of the top color
3. Rotate the device

Most likely, you will find that the top color is different than before. This implies that we generated a new random set of colors. And, if you keep switching the device between portrait and landscape, you would keep getting new colors.

That is not great.

It would be worse if our data were coming from disk, or from the network, instead of being randomly generated. If we have to re-load our data every time that the user rotates the screen, we would wind up having to make many extra requests of our database (local or remote), wasting time and battery.

In this chapter, we will focus on how to avoid this problem, through a viewmodel.

Configuration Changes

When you call methods in the Android SDK that load a resource (e.g., `setContentView(R.layout.activity_main)`), Android will walk through your 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 might want a version of our layouts that can take advantage of the wider screen, 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.

The biggest thing that Android does on a configuration change 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.

What We Want... and What We Do Not Want

We want to present a user interface that fits well with the current configuration. For example, we want to take advantage of the screen space that is given to us.

Conversely, we do not want to have a UI that fits poorly in the given space. For example, in landscape mode, our layout might be too tall. Even if we use a `ScrollView` to make it vertically scrollable, so the user can take steps to see the entire layout, that may not be very user-friendly.

However, despite the UI change, we want the user to think that nothing much unusual happened when the user triggered the configuration.

We do **not** want the user to regret that configuration change, such as by losing the data that we were showing (or, worse, that the user had entered or changed).

Over the years, we have spent a fair amount of time dealing with this challenge. There is a two-tier solution that seems to work best. In this chapter, we will focus on

one of those tiers: retaining data using `ViewModel`.

Enter the `ViewModel`

So, a configuration change destroys and recreates our activity, by default. As a result, anything that is held onto uniquely by an activity instance — such as our random numbers — gets lost when we switch to the new activity instance, by default.

What would be nice is if we could separate our activity data into two groups:

- Part of our data, such as references to widgets, can be discarded on a configuration change, as we will need to get fresh data in the new activity
- Part of our data, such as our random numbers, could be passed from the old activity instance to the new activity instance, so we can present that data again to the user

The Jetpack solution for this is `ViewModel`. We can create a subclass of `ViewModel` that holds onto the second category of activity data: the stuff that we want to reuse after the configuration change. Jetpack will take the steps necessary to get that data — indeed, typically the entire `ViewModel` object — from the old activity instance to the new one. All that we need to do is make intelligent choices about what goes into the `ViewModel` and what does not.

Applying `ViewModel`

The `ViewModel` sample module in the [Sampler](#) and [SamplerJ](#) projects start with the same code that we used in `RecyclerViewBasics`. So, we have a `RecyclerView` in an activity that is set up to display 25 random numbers as colors. The difference is that this time, we will use a `ViewModel` to retain those random numbers across a configuration change. Also, we will log the lifecycle methods, so we can see them as they are executed.

The Dependencies

You get `ViewModel` via a transitive dependency from `androidx.appcompat:appcompat` — Java developers do not need anything else.

Kotlin developers will want to add `androidx.lifecycle:lifecycle-viewmodel-ktx` as a dependency. This adds a few Kotlin-specific extension functions associated with

INTEGRATING VIEWMODEL

ViewModel and related classes. They make using ViewModel a little bit easier in Kotlin than if you were to use the Java API alone. This is a common pattern in Jetpack: have a Java API with a separate library containing Kotlin extension functions. The “Android KTX” brand is used for these Kotlin extension function libraries.

Kotlin developers also will want to add `androidx.activity:activity-ktx` as a dependency. This adds some Kotlin-specific extension functions to Activity, including ones that we will use when working with ViewModel.

So, in our Kotlin project, we have:

```
dependencies {  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.recyclerview:recyclerview:1.2.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.1'  
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'  
    implementation "androidx.activity:activity-ktx:1.3.1"  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
}
```

(from [ViewModel/build.gradle](#))

The Java project just needs appcompat:

```
dependencies {  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.recyclerview:recyclerview:1.2.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.1'  
}
```

(from [ViewModel/build.gradle](#))

The ViewModel

Our project now has a `ColorViewModel` class that inherits from Jetpack’s `ViewModel`, both in Java:

```
package com.commonware.jetpack.samplerj.viewmodel;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
import androidx.lifecycle.ViewModel;
```

INTEGRATING VIEWMODEL

```
public class ColorViewModel extends ViewModel {
    private final Random random = new Random();
    final List<Integer> numbers = buildItems();

    private List<Integer> buildItems() {
        ArrayList<Integer> result = new ArrayList<>(25);

        for (int i = 0; i < 25; i++) {
            result.add(random.nextInt());
        }

        return result;
    }
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/ColorViewModel.java](#))

...and Kotlin:

```
package com.commonsware.jetpack.sampler.viewmodel

import androidx.lifecycle.ViewModel
import java.util.*

class ColorViewModel : ViewModel() {
    private val random = Random()
    val numbers = List(25) { random.nextInt() }
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/ColorViewModel.kt](#))

Our random numbers are now created in the ColorViewModel, via property initializers. Otherwise, our ColorViewModel itself is fairly unremarkable.

Using the ViewModel

How you get a ViewModel depends a bit on whether you are using Java or Kotlin.

Java

Our activity can get its instance of ColorViewModel from a ViewModelProvider, which we can create using an ordinary constructor. On that, we can call get() to retrieve our ColorViewModel. We need to pass the Java Class object for our ColorViewModel as a parameter to get():

INTEGRATING VIEWMODEL

```
ColorViewModel vm = new ViewModelProvider(this).get(ColorViewModel.class);
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/MainActivity.java](#))

We can then use the numbers property of our ColorViewModel to populate our RecyclerView:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    final ActivityMainBinding binding =
        ActivityMainBinding.inflate(getLayoutInflater());

    setContentView(binding.getRoot());

    ColorAdapter adapter = new ColorAdapter(getLayoutInflater());
    ColorViewModel vm = new ViewModelProvider(this).get(ColorViewModel.class);

    adapter.submitList(vm.numbers);
    binding.items.setLayoutManager(new LinearLayoutManager(this));
    binding.items.addItemDecoration(
        new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
    binding.items.setAdapter(adapter);

    Log.d(TAG, "onCreate() called!");
}
```

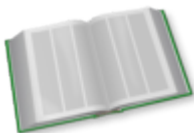
(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/MainActivity.java](#))

Kotlin

In Kotlin, the activity-ktx library gives us a viewModels property delegate that we can use:

```
val vm: ColorViewModel by viewModels()
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/MainActivity.kt](#))



You can learn more about property delegates in the "Property Delegates" chapter of [Elements of Kotlin](#)!

Then, as with Java, we can use the numbers property of our ColorViewModel to

populate our RecyclerView:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityMainBinding.inflate(layoutInflater)

    setContentView(binding.root)

    val vm: ColorViewModel by viewModels()

    binding.items.apply {
        layoutManager = LinearLayoutManager(this@MainActivity)
        addItemDecoration(
            DividerItemDecoration(
                this@MainActivity,
                DividerItemDecoration.VERTICAL
            )
        )
        adapter = ColorAdapter(layoutInflater).apply {
            submitList(vm.numbers)
        }
    }

    Log.d(TAG, "onCreate() called!")
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/MainActivity.kt](#))

The Older Solution

Earlier versions of this book, and various blog posts and articles, will show `ViewModelProviders.of()` as the way to get a `ViewModel`. This was deprecated and should no longer be used, as it may be removed in future versions of the `lifecycle-extensions` library.

The Results

If you run this sample app, and you rotate the screen, the random numbers are the same.

The reason this works is that we have the *same* `ColorViewModel` instance in both the old activity (before the configuration change) and the new activity (after the configuration change). `ViewModelProvider` and the rest of the `ViewModel` portion of

Jetpack will ensure that we have the same `ColorViewModel` in the old and new activity instances. Since our `ColorViewModel` holds our data, our data is retained across the configuration change, so we have the same numbers in both the old and the new activity.

ViewModel and the Lifecycle

Our `MainActivity` also has calls to `Log.d()` in the main lifecycle functions, both in Java:

```
package com.commonware.jetpack.samplerj.viewmodel;

import android.os.Bundle;
import android.util.Log;
import com.commonware.jetpack.samplerj.viewmodel.databinding.ActivityMainBinding;
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProvider;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "ViewModel";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final ActivityMainBinding binding =
            ActivityMainBinding.inflate(getLayoutInflater());

        setContentView(binding.getRoot());

        ColorAdapter adapter = new ColorAdapter(getLayoutInflater());
        ColorViewModel vm = new ViewModelProvider(this).get(ColorViewModel.class);

        adapter.submitList(vm.numbers);
        binding.items.setLayoutManager(new LinearLayoutManager(this));
        binding.items.addItemDecoration(
            new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
        binding.items.setAdapter(adapter);

        Log.d(TAG, "onCreate() called!");
    }

    @Override
    protected void onStart() {
        super.onStart();

        Log.d(TAG, "onStart() called!");
    }

    @Override
    protected void onResume() {
        super.onResume();
    }
}
```

```
    Log.d(TAG, "onResume() called!");
}

@Override
protected void onPause() {
    Log.d(TAG, "onPause() called!");

    super.onPause();
}

@Override
protected void onStop() {
    Log.d(TAG, "onStop() called!");

    super.onStop();
}

@Override
protected void onDestroy() {
    Log.d(TAG, "onDestroy() called!");

    super.onDestroy();
}
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/MainActivity.java](#))

...and in Kotlin:

```
package com.commonsware.jetpack.sampler.viewmodel

import android.os.Bundle
import android.util.Log
import androidx.activity.viewModels
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.jetpack.sampler.viewmodel.databinding.ActivityMainBinding

private const val TAG = "ViewModel"

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)

        val vm: ColorViewModel by viewModels()

        binding.items.apply {
            layoutManager = LinearLayoutManager(this@MainActivity)
            addItemDecoration(
                DividerItemDecoration(
                    this@MainActivity,
                    DividerItemDecoration.VERTICAL
                )
            )
        }
    }
}
```

```
    )
    adapter = ColorAdapter(layoutInflater).apply {
        submitList(vm.numbers)
    }
}

Log.d(TAG, "onCreate() called!")
}

override fun onStart() {
    super.onStart()

    Log.d(TAG, "onStart() called!")
}

override fun onResume() {
    super.onResume()

    Log.d(TAG, "onResume() called!")
}

override fun onPause() {
    Log.d(TAG, "onPause() called!")

    super.onPause()
}

override fun onStop() {
    Log.d(TAG, "onStop() called!")

    super.onStop()
}

override fun onDestroy() {
    Log.d(TAG, "onDestroy() called!")

    super.onDestroy()
}
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/MainActivity.kt](#))

As you run the app, and as you undergo configuration changes, you will see the lifecycle methods get logged, by looking for these messages in the Logcat view in Android Studio. For example, starting the app will result in Logcat messages like:

```
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
```

If you rotate the screen, the list of messages becomes:

```
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
```

INTEGRATING VIEWMODEL

```
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onPause() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onStop() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onDestroy() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
```

If you press BACK to exit the rotated activity, the list of messages becomes:

```
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onPause() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onStop() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onDestroy() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onPause() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onStop() called!  
com.commonware.jetpack.sampler.viewmodel D/ViewModel: onDestroy() called!
```

We are seeing our original activity instance come onto the screen, then get destroyed and a replacement created, before it too gets destroyed as the user exits the activity.

Our `ColorViewModel` is oblivious to most of this. It will get created as part of the original `onCreate()` call. When the new activity instance is in its `onCreate()`, and it attempts to retrieve the `ColorViewModel`, Jetpack realizes that we already have a `ColorViewModel` from the previous activity instance, so it reuses it.

If we wanted, we could override `onCleared()` in `ColorViewModel`. This will be called when our activity is “really” destroyed:

- `onDestroy()` is called, and
- We are not undergoing a configuration change

In `onCleared()`, we could clean up anything that we needed to clean up. Often, there is nothing that you need to clean up specifically — after all, the `ViewModel` will get garbage-collected shortly. However, sometimes your `ViewModel` might hold onto things that should get cleaned up, such as:

- References to threads, Kotlin coroutines, or the like
- Open Internet connections to some server

- Open references to local databases
- And so on

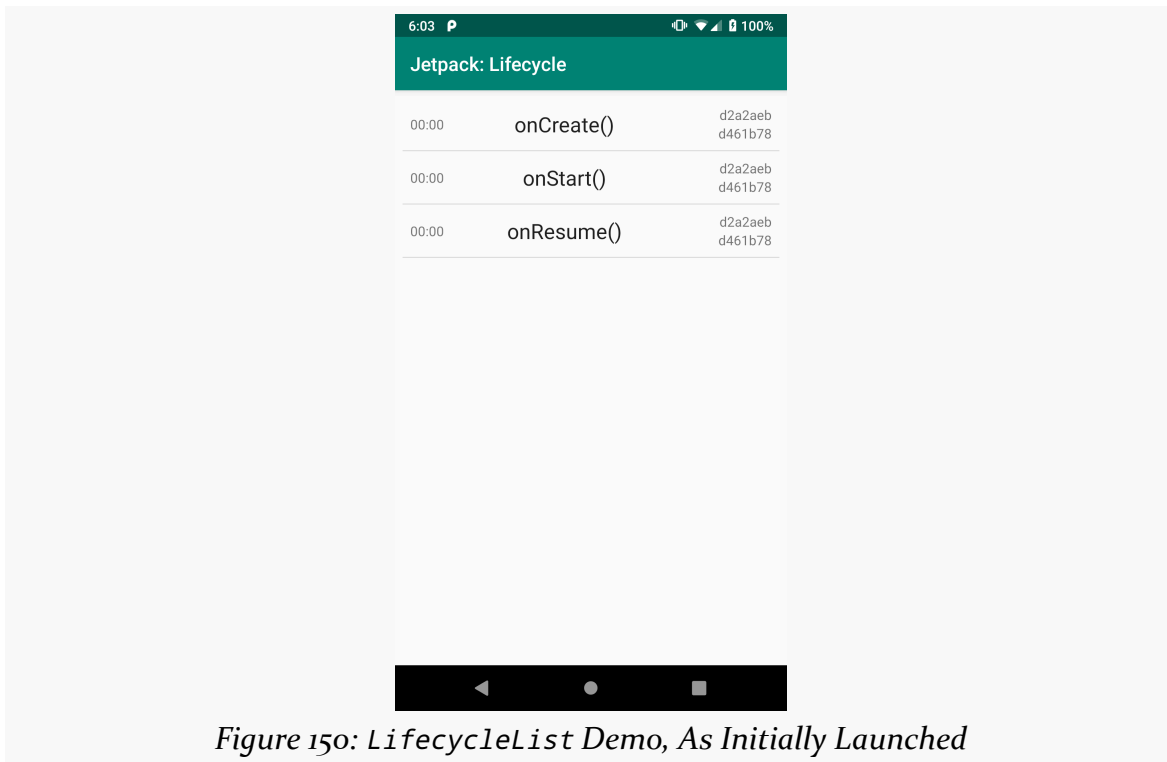
To the extent that your `ViewModel` has those, `onCleared()` is where you would release those things, such as disconnecting from the server.

Changing Data in the ViewModel

In our 25-random-numbers example, the data never changed. That is what we wanted for that example, so our random numbers would be consistent through configuration changes.

However, it is not very realistic. Most of the time, our data changes when the user uses the app, and we need our viewmodel to handle that.

The `LifecycleList` sample module in the [Sampler](#) and [SamplerJ](#) projects blend our lifecycle-logging logic with our show-a-list logic. Now, instead of showing 25 random colors, we will show a list of lifecycle events, collected as the user uses the app:



Each row in the list now shows four pieces of data:

- The lifecycle method that was called (e.g., onCreate())
- The number of seconds since the app was started when the event occurred (e.g., 0:00 for initial events)
- A random number identifying the activity (in the upper right corner)
- A random number identifying the viewmodel (in the lower right corner)

These latter two values will help show us when we wind up with different instances of those objects.

The Event Model

We need some object to hold that data to be shown in each of our RecyclerView rows. So, this project has an Event model class, in Java:

```
package com.commonware.jetpack.samplerj.lifecycle;

import android.os.SystemClock;

class Event {
    final long timestamp = SystemClock.elapsedRealtime();
    final String message;
    final int activityHash;
    final int viewmodelHash;

    Event(String message, int activityHash, int viewmodelHash) {
        this.message = message;
        this.activityHash = activityHash;
        this.viewmodelHash = viewmodelHash;
    }
}
```

(from [LifecycleList/src/main/java/com/commonware/jetpack/samplerj/lifecycle/Event.java](#))

...or Kotlin:

```
package com.commonware.jetpack.sampler.lifecycle

import android.os.SystemClock

data class Event(
    val message: String,
    val activityHash: Int,
```

```
val viewModelHash: Int,  
val timestamp: Long = SystemClock.elapsedRealtime()  
)
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/Event.kt](#))

The two “hash code” values are Int properties. The lifecycle method is a String referred to as the message. And we have a timestamp property that will track when this Event was created. For that, we use `SystemClock.elapsedRealtime()`, which returns the number of milliseconds since the device was powered on.

The New RecyclerView Bits

Our row layout now needs widgets for those four pieces of data that we wish to display:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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="wrap_content"  
    android:padding="@dimen/content_padding">  
  
    <TextView  
        android:id="@+id/activityHash"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintTop_toTopOf="parent"  
        tools:text="0x12345678" />  
  
    <TextView  
        android:id="@+id/viewmodelHash"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        app:layout_constraintBottom_toBottomOf="parent"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintTop_toBottomOf="@id/activityHash"  
        tools:text="0x90ABCDEF" />  
  
    <androidx.constraintlayout.widget.Barrier  
        android:id="@+id/barrier"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content">
```

```

        android:layout_marginEnd="4dp"
        android:layout_marginStart="4dp"
        app:barrierDirection="start"
        app:constraint_referenced_ids="activityHash,viewModelHash" />

<TextView
    android:id="@+id/timestamp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="01:23" />

<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@id/barrier"
    app:layout_constraintStart_toEndOf="@id/timestamp"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

(from [LifecycleList/src/main/res/layout/row.xml](#))

Each of the four pieces of data is represented by a `TextView`. The two hash values are anchored to the “end” side of the `ConstraintLayout`, with the `activityHash` on the top and the `viewModelHash` on the bottom. The timestamp widget is anchored on the “start” side of the `ConstraintLayout`, centered between the top and the bottom. And the message widget is centered in the remaining space, using a `Barrier` to determine where the hashes start.

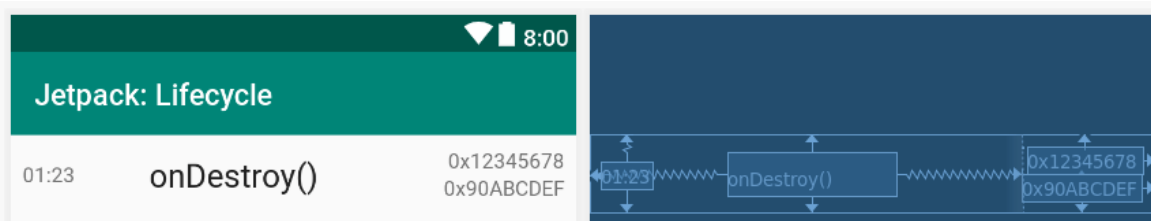


Figure 151: Android Studio Layout Editor, Showing row Layout

INTEGRATING VIEWMODEL

Note that we dropped the `android:background`, `android:clickable`, and `android:focusable` attributes from the `ConstraintLayout`, as this particular sample is not going to respond to click events on the rows. That allows our new view-holder class, `EventViewHolder`, to just focus on populating the widgets for its row:

```
package com.commonware.jetpack.samplerj.lifecycle;

import android.text.format.DateUtils;
import com.commonware.jetpack.samplerj.lifecycle.databinding.RowBinding;
import androidx.recyclerview.widget.RecyclerView;

class EventViewHolder extends RecyclerView.ViewHolder {
    private final RowBinding binding;
    private final long startTime;

    EventViewHolder(RowBinding binding, long startTime) {
        super(binding.getRoot());

        this.binding = binding;
        this.startTime = startTime;
    }

    void bindTo(Event event) {
        long elapsedSeconds = (event.timestamp - startTime)/1000;

        binding.timestamp.setText(DateUtils.formatElapsedTime(elapsedSeconds));
        binding.message.setText(event.message);
        binding.activityHash.setText(Integer.toHexString(event.activityHash));
        binding.viewmodelHash.setText(Integer.toHexString(event.viewmodelHash));
    }
}
```

(from [LifecycleList/src/main/java/com/commonware/jetpack/samplerj/lifecycle/EventViewHolder.java](#))

```
package com.commonware.jetpack.sampler.lifecycle

import android.text.format.DateUtils
import androidx.recyclerview.widget.RecyclerView
import com.commonware.jetpack.sampler.lifecycle.databinding.RowBinding

class EventViewHolder(
    private val binding: RowBinding,
    private val startTime: Long
) : RecyclerView.ViewHolder(binding.root) {
    fun bindTo(event: Event) {
        val elapsedSeconds = (event.timestamp - startTime) / 1000
    }
}
```

INTEGRATING VIEWMODEL

```
binding.timestamp.text = DateUtils.formatElapsedTime(elapsedSeconds)
binding.message.text = event.message
binding.activityHash.text = Integer.toHexString(event.activityHash)
binding.viewmodelHash.text = Integer.toHexString(event.viewmodelHash)
}
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/EventViewHolder.kt](#))

For the timestamp, we use `DateUtils.formatElapsedTime()`. This is a utility method provided by Android that formats a number of seconds into an HH:MM:SS format to show the elapsed time in hours, minutes, and seconds. Note that the Event value for the timestamp is in milliseconds, as is the `startTime` value that is being passed into `EventViewHolder`, so we need to divide our net time by 1000 to convert the milliseconds to seconds.

Our new adapter — `EventAdapter` — wraps a List of Event objects and pours them into `EventViewHolder` objects as needed:

```
package com.commonsware.jetpack.samplerj.lifecycle;

import android.os.SystemClock;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.commonsware.jetpack.samplerj.lifecycle.databinding.RowBinding;
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.DiffUtil;
import androidx.recyclerview.widget.ListAdapter;

class EventAdapter extends ListAdapter<Event, EventViewHolder> {
    private final LayoutInflater inflater;
    private final long startTime;

    EventAdapter(LayoutInflater inflater, long startTime) {
        super(DIFF_CALLBACK);
        this.inflater = inflater;
        this.startTime = startTime;
    }

    @NonNull
    @Override
    public EventViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                           int viewType) {
        final RowBinding binding = RowBinding.inflate(inflater, parent, false);

        return new EventViewHolder(binding, startTime);
    }

    @Override
    public void onBindViewHolder(@NonNull EventViewHolder holder, int position) {
        holder.bindTo(getItem(position));
    }
}
```

INTEGRATING VIEWMODEL

```
private static final DiffUtil.ItemCallback<Event> DIFF_CALLBACK =
    new DiffUtil.ItemCallback<Event>() {
        @Override
        public boolean areItemsTheSame(@NonNull Event oldEvent, @NonNull Event newEvent) {
            return oldEvent == newEvent;
        }

        @Override
        public boolean areContentsTheSame(@NonNull Event oldEvent, @NonNull Event newEvent) {
            return oldEvent.timestamp == newEvent.timestamp &&
                oldEvent.message.equals(newEvent.message) &&
                oldEvent.activityHash == newEvent.activityHash &&
                oldEvent.viewmodelHash == newEvent.viewmodelHash;
        }
    };
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/samplerj/lifecycle/EventAdapter.java](#))

```
package com.commonsware.jetpack.sampler.lifecycle

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import com.commonsware.jetpack.sampler.lifecycle.databinding.RowBinding

internal class EventAdapter(
    private val inflater: LayoutInflater,
    private val startTime: Long
) : ListAdapter<Event, EventViewHolder>(EventDiffer) {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ) = EventViewHolder(RowBinding.inflate(inflater, parent, false), startTime)

    override fun onBindViewHolder(holder: EventViewHolder, position: Int) {
        holder.bindTo(getItem(position))
    }

    private object EventDiffer : DiffUtil.ItemCallback<Event>() {
        override fun areItemsTheSame(oldEvent: Event, newEvent: Event) =
            oldEvent === newEvent

        override fun areContentsTheSame(oldEvent: Event, newEvent: Event) =
            oldEvent == newEvent
    }
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/EventAdapter.kt](#))

For our `DiffUtil.ItemCallback` implementation, we use identity equality to determine whether the items are the same and content equality to determine if the contents are the same. Here, though, we have more substantial language differences:

- In Java, identity equality is via `==`, and we have to examine each one of the `Event` fields ourselves in `areContentsTheSame()`
- In Kotlin, identity is via `===`, and we can use object equality (`==`) for our `Event`, since it is a data class and generates an `equals()` function that compares each property for us

The EventViewModel

Our `ViewModel` is now called `EventViewModel`, and it has two properties:

- `events`, which is a list of the events that we have had to date
- `startTime`, which is the time when the `EventViewModel` is created, once again obtained via `SystemClock.elapsedRealtime()`

```
package com.commonware.jetpack.samplerj.lifecycle;

import android.os.SystemClock;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import androidx.lifecycle.ViewModel;

public class EventViewModel extends ViewModel {
    final List<Event> events = new ArrayList<>();
    final long startTime = SystemClock.elapsedRealtime();
    private final int id = new Random().nextInt();

    void addEvent(String message, int activityHash) {
        events.add(new Event(message, activityHash, id));
    }

    @Override
    protected void onCleared() {
        events.clear();
    }
}
```

(from [LifecycleList/src/main/java/com/commonware/jetpack/samplerj/lifecycle/EventViewModel.java](#))

```
package com.commonware.jetpack.sampler.lifecycle

import android.os.SystemClock
import androidx.lifecycle.ViewModel
import java.util.*

class EventViewModel : ViewModel() {
    val events: MutableList<Event> = mutableListOf()
    val startTime = SystemClock.elapsedRealtime()
    private val id = Random().nextInt()

    fun addEvent(message: String, activityHash: Int) {
        events.add(Event(message, activityHash, id))
    }

    override fun onCleared() {
        events.clear()
    }
}
```

(from [LifecycleList/src/main/java/com/commonware/jetpack/sampler/lifecycle/EventViewModel.kt](#))

We also have:

- `addEvent()`, to record an event when it occurs
- `onCleared()`, to clear the events list when the `EventViewModel` is no longer being used

The `onCleared()` implementation is unnecessary, as the events list would be garbage-collected when the `EventViewModel` is. We have it here just to show you what overriding that method looks like.

Updating the EventViewModel

`MainActivity` now has an `addEvent()` function to update the `EventViewModel` when lifecycle events occur:

```
package com.commonware.jetpack.samplerj.lifecycle;

import android.os.Bundle;
import com.commonware.jetpack.samplerj.lifecycle.databinding.ActivityMainBinding;
import java.util.ArrayList;
import java.util.Random;
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProvider;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;
```

INTEGRATING VIEWMODEL

```
public class MainActivity extends AppCompatActivity {
    private EventAdapter adapter;
    private EventViewModel vm;
    private final int id = new Random().nextInt();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final ActivityMainBinding binding =
            ActivityMainBinding.inflate(getLayoutInflater());

        setContentView(binding.getRoot());

        vm = new ViewModelProvider(this).get(EventViewModel.class);
        adapter = new EventAdapter(getLayoutInflater(), vm.startTime);
        addEvent("onCreate()");

        binding.items.setLayoutManager(new LinearLayoutManager(this));
        binding.items.addItemDecoration(
            new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
        binding.items.setAdapter(adapter);
    }

    @Override
    protected void onStart() {
        super.onStart();

        addEvent("onStart()");
    }

    @Override
    protected void onResume() {
        super.onResume();

        addEvent("onResume()");
    }

    @Override
    protected void onPause() {
        addEvent("onPause()");

        super.onPause();
    }

    @Override
    protected void onStop() {
        addEvent("onStop()");

        super.onStop();
    }

    @Override
    protected void onDestroy() {
        addEvent("onDestroy()");

        super.onDestroy();
    }

    private void addEvent(String message) {
```

INTEGRATING VIEWMODEL

```
        vm.addEvent(message, id);
        adapter.submitList(new ArrayList<>(vm.events));
    }
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/samplerj/lifecycle/MainActivity.java](#))

```
package com.commonsware.jetpack.sampler.lifecycle

import android.os.Bundle
import androidx.activity.viewModels
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.jetpack.sampler.lifecycle.databinding.ActivityMainBinding
import java.util.*
import kotlin.collections.ArrayList

class MainActivity : AppCompatActivity() {
    private lateinit var adapter: EventAdapter
    private val vm: EventViewModel by viewModels()
    private val id = Random().nextInt()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)

        adapter = EventAdapter(layoutInflater, vm.startTime)
        addEvent("onCreate()")

        binding.items.layoutManager = LinearLayoutManager(this)
        binding.items.addItemDecoration(
            DividerItemDecoration(this, DividerItemDecoration.VERTICAL)
        )
        binding.items.adapter = adapter
    }

    override fun onStart() {
        super.onStart()

        addEvent("onStart()")
    }

    override fun onResume() {
        super.onResume()

        addEvent("onResume()")
    }

    override fun onPause() {
        addEvent("onPause()")

        super.onPause()
    }

    override fun onStop() {
        addEvent("onStop()")
    }
}
```

```
    super.onStop()
}

override fun onDestroy() {
    addEvent("onDestroy()")

    super.onDestroy()
}

private fun addEvent(message: String) {
    vm.addEvent(message, id)
    adapter.submitList(ArrayList(vm.events))
}
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/MainActivity.kt](#))

`addEvent()` also calls `submitList()` on our `EventAdapter`, to update the `RecyclerView` when we add events to the list. This requires us to hold onto the `EventAdapter` and the `EventViewModel` in properties of the activity, rather than just using them as local variables in `onCreate()`.

When `addEvent()` calls `submitList()`, it creates a new `ArrayList` from the events. That is because `ListAdapter` and `submitList()` will “short-circuit” the update logic if you pass the same `List` to `submitList()` that you did the previous call. `ListAdapter` assumes that nothing needs to be done in that case, as `ListAdapter` assumes that the list *contents* did not change. To get past this, we have to pass in a brand-new `ArrayList` object each time.

The Results

When you run the app, lifecycle events show up in the list. If you rotate the screen a couple of times, the events start to pile up:



The activity “hash code” value in the upper right changes with each configuration change, as we get a fresh instance of MainActivity each time. However, the viewmodel “hash code” value in the lower right remains the same, as we are using the same EventViewModel instance for each of our MainActivity instances.

ViewModel and AndroidViewModel

Sometimes, your ViewModel could really use a Context. For example, you might need to format some data using a string resource, and the only way to get a string resource is through a Context.

If you need a Context in your ViewModel, Google’s official solution is AndroidViewModel. This offers a `getApplication()` function that you can call to retrieve the Application singleton. Application implements Context, so you can

look up string resources and stuff from there. Your activity or fragment does not need to worry about this: just use `ViewModelProvider` and request the `ViewModel`. You also will extend `AndroidViewModel` instead of `ViewModel` and implement the required constructor, that takes an `Application` as a parameter.

Some of the examples in the upcoming chapters will use `ViewModel`, while others will use `AndroidViewModel`.

ViewModelFactory

A `ViewModelProvider` will use a `ViewModelProvider.Factory` to create instances of your `ViewModel` subclass. If you have no constructor parameters — or if you are extending `AndroidViewModel` and just have the `Application` constructor parameter — then the default `ViewModelProvider.Factory` will suffice. No special configuration is necessary.

However, there will be times when you will want to pass other constructor parameters to your `ViewModel` subclass. You can have those parameters, but now the Jetpack does not know how to create instances of your `ViewModel`. You will need to supply the `ViewModelProvider` a `ViewModelProvider.Factory` that knows how to create those instances.

Understanding Processes

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.

And, along the way, we will answer nagging questions like, “what is that `Bundle` thing that we are passed in `onCreate()` that we have been ignoring?”.

When Processes Are Created

A user installs your app, goes to their launcher’s app drawer, and taps on an icon representing your activity. Your activity dutifully appears on the screen.

At this point, we know that we have a process for our app, because it is running.

Frequently, the trigger for Android to fork a process for you is that the user tapped on your launcher icon and launched your activity. However, your process might already exist for other reasons. In that case, by default, Android will reuse your existing process, and have it display your activity. By default, your app will have at most one process.

What Is In Your Process

When Android needs a process for your app — such as to show the launcher activity — Android forks 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 a virtual machine for running your app, 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 shared native libraries, such as for SSL encryption and local database access, also shared via copy-on-write memory
- A copy of your own classes, loaded out of your APK
- A copy of classes from libraries in your app, loaded out of your APK
- Any native libraries (e.g., written in C/C++) that you linked into your app, loaded out of your APK
- Any objects created by you or the framework classes, such as the instance of your `Activity` subclass

So, our process has a lot of things in it. Much of it is shared among all other Android SDK apps forked from the zygote. The unique elements will be those things that we use from our APK.

BACK, HOME, and Your Process

So far, our sample apps have had just one activity. So, imagine this scenario: from the app drawer, 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. Our activity is destroyed. Control will return to whatever preceded that activity — in this case, the 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, frequently there is little difference: the launcher re-appears.

The difference is what happens to your activity.

When the user presses BACK, your foreground activity is destroyed. It will be called with `onDestroy()` among the other teardown lifecycle methods. And 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 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 launcher's own activity back to the foreground. Otherwise, it does not affect your process very 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 apps and their processes.

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 launcher, where the user can return to the launcher 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”.

There is an element of the Android OS called the “out-of-memory killer”. Its job is to ensure that there is a reasonable amount of free system RAM available at all times, so Android can fork new processes for apps when needed. When free system RAM drops below a certain level, the out-of-memory killer will terminate a process. The out-of-memory killer will tend to terminate empty processes more readily than it will terminate process with 1+ running components. However, the out-of-memory killer also takes into account things like:

- Process age, where older processes will be somewhat more likely to be terminated
- Memory usage, where bigger processes will be somewhat more likely to be terminated

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.

Tasks and Your App

If you have used an Android device for a reasonable length of time, most likely you will have seen the “overview screen, sometimes called the “recent-tasks list”. The look has varied widely over the years and across the range of devices and manufacturers. They all give you a way to [“get back to where you once belonged”](#) — you can rapidly return to some app that you had been in recently.

So, on Android 9.0 you might see:

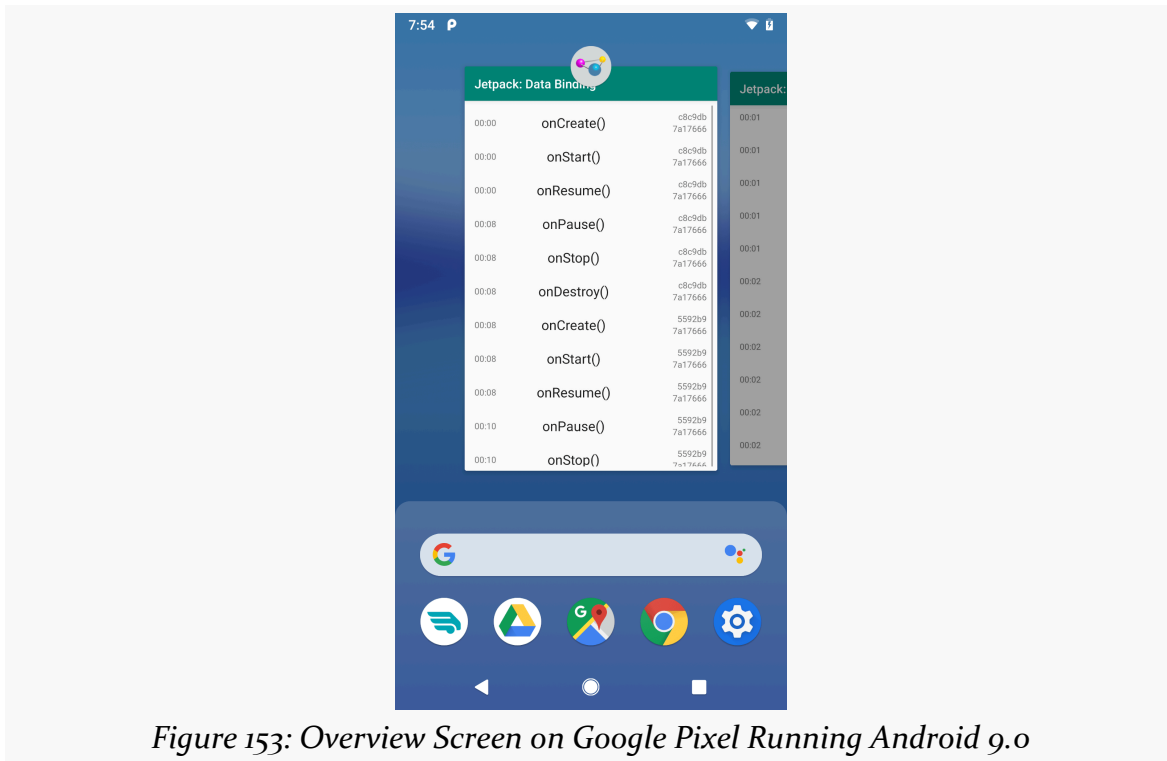


Figure 153: Overview Screen on Google Pixel Running Android 9.0

...while on Android 5.0, you might see:

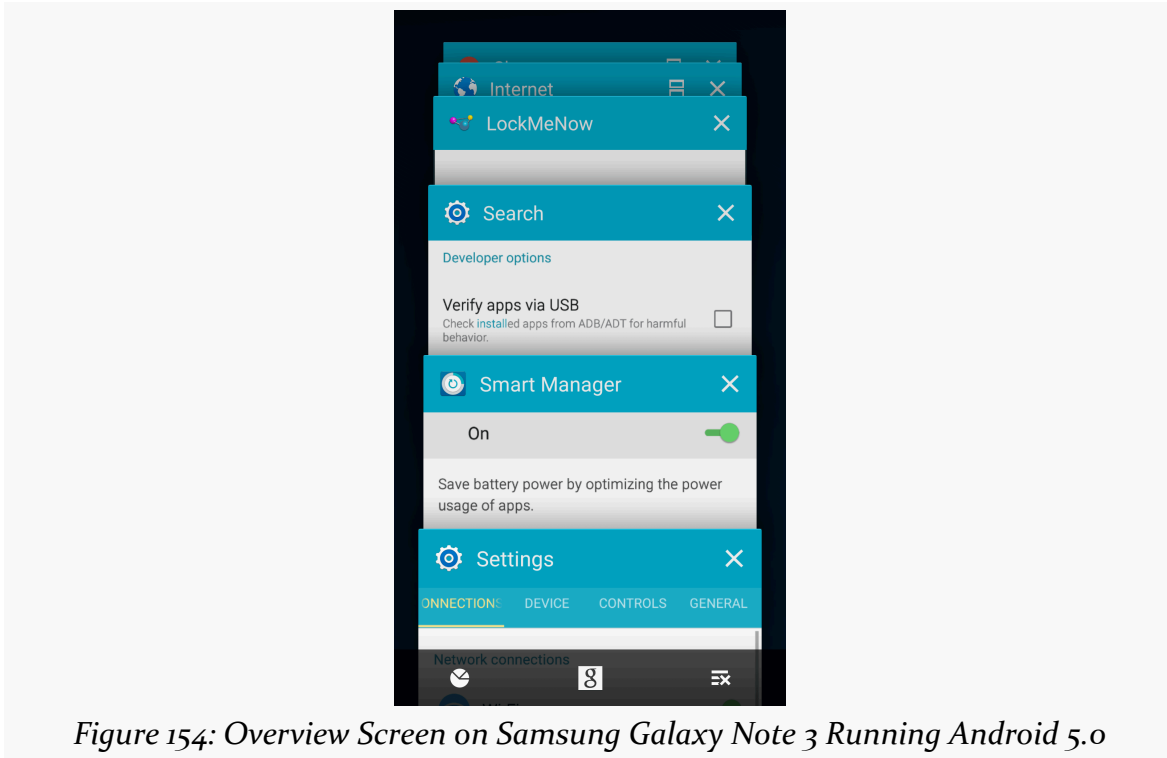


Figure 154: Overview Screen on Samsung Galaxy Note 3 Running Android 5.0

Tasks are the sort of thing that developers often ignore early on. However, many apps eventually do need to address task management.

What is a Task?

Tasks are a bit like tabs in a Web browser.

In a tabbed Web browser, each tab operates fairly independently from other tabs. In particular, each tab has its own “back stack”: pressing the BACK button affects your current tab, not other tabs that happen to be open.

Similarly, a task in Android represents a stack of activities. One activity can open other activities, in the same app or even in different apps, as we will explore [in an upcoming chapter](#). By default, those activities will all belong to the same task. Each time the user presses the BACK button, it destroys the current foreground activity and returns control to whatever activity preceded it. If the user presses BACK from the first (“root”) activity of the task, the task itself is destroyed and the user returns to the home screen.

OK, So Why Do I Care?

The point of tasks is to allow users to rapidly move between apps that have been used recently. In principle, this is the same as a desktop operating system allowing the user to move between opened apps.

However, there is one practical difference: mobile devices usually have a lot less RAM. While newer high-end Android devices might have a lot of RAM, older or less-expensive Android devices might not have very much. As a result, we cannot have lots of running processes — the out-of-memory killer will eventually have to start killing off those processes to free up memory for yet other processes.

However, once an app's process is terminated, everything it held onto in memory is now gone. It has to reload data from disk or from the network. Moreover, it loses all “context” by default: the app does not necessarily remember where the user had been in the app. Instead, by default, if the user goes back to the app, the app will start over from scratch.

To help deal with this, Android keeps track of some amount of state associated with your app. If:

- The user navigates away from your app,
- Android terminates your app's process, and
- The user attempts to return to your app within 30 minutes of having left it

...then Android will try to restore your app and its task to the state it had been in when the user left. In particular, for the purposes of this chapter, Android will hand some “instance state” back to your activity, and you can try to use that to pretend to the user that your activity had been around all along, even though in reality this is a brand-new activity instance in a brand-new process.

Instance State

So, with that in mind, let's talk a bit about “instance state”.

Why Are We Passed a Bundle in onCreate()?

You may have noticed that all of our activities in the sample apps so far have an `onCreate()` function that takes a `Bundle` as a parameter. Each of the samples has done work in `onCreate()`... and each of the samples has ignored this `Bundle`.

This Bundle is known as the “saved instance state”.

A Bundle is a key-value store, using strings for keys and a limited set of data types for the values. Mostly, a Bundle is designed to hold simple bits of data: Boolean, Int, Long, String, etc.

Normally, when our activity is created, the Bundle parameter to `onCreate()` is null. That is why in Kotlin it is typed as `Bundle?` instead of `Bundle`.

If that Bundle is not null, it represents some state from some previous instance of our activity, where that previous instance either:

- Had been destroyed as part of a configuration change, or
- Had been nuked from orbit as part of Android terminating the app’s process, but the user happened to try returning to that app within 30 minutes of having left it

Our job is to use that Bundle to help set up our user interface.

Nowadays, for the configuration-change scenario, we usually use a `ViewModel` or something like that. However, a `ViewModel` also gets nuked from orbit when the process is terminated, so that does not help us for the “return in 30 minutes” scenario.

When Do We Fill In the Instance State Bundle?

Your activity can optionally override an `onSaveInstanceState()` function. This will be passed a Bundle, and you can fill data into that Bundle.

Typically, you chain to the superclass, as `Activity` will put things into the Bundle related to your current on-screen widgets. In particular, `Activity` will save some user-mutable data from those widgets, such as:

- The text that the user has typed into an `EditText`
- The checked states of `CompoundButton` widgets like `Switch` or `CheckBox`
- The position of a `SeekBar`

When you chain to the superclass implementation of `onSaveInstanceState()`, `Activity` will save all that data for you. In addition, you can put your own data into the Bundle.

In current versions of the Jetpack, we can adapt our `ViewModel` classes to handle this work for us. We will see that [later in the chapter](#).

When Do We Get the Instance State Bundle?

Your activity will get a copy of that Bundle back in two places:

- `onCreate()`, and
- `onRestoreInstanceState()`

`onCreate()` is always called for a new activity instance. However, there may not be any instance state to restore, so its Bundle parameter may be `null`.

`onRestoreInstanceState()` is only called if there is instance state to restore, so its Bundle will never be `null`.

Your job is to look for this Bundle and apply any data from it that you put into the Bundle in `onSaveInstanceState()`. Or, [let your ViewModel handle that](#).

What Are the Limits on the Bundle?

As noted above, Bundle does not support arbitrary data types. For your own custom classes, you can address this limitation via `Parcelable`, as we will see [in the next section](#). But you should not assume that you can put any sort of data into the Bundle.

There is also a size limitation. The details are rather complicated, but the general rule of thumb is “keep your Bundle *well* under 1MB”.

The instance state Bundle is not designed to be a replacement for a local database or a server. Use it for in-flight data that you cannot easily load again and that should fit the data type and size limitations. Use your `ViewModel` for data that you have loaded from disk or the network that you would prefer not to load again after a configuration change, and live with the fact that you *will* need to load that data again after your process has been terminated.

Pondering Parcelable

In the `LifecycleList` sample app from [the previous chapter](#), our `EventViewModel` held onto two pieces of data:

- The time when the `EventViewModel` was created, and
- The list of Event objects representing the lifecycle events

If our process is terminated, but the user returns to us with 30 minutes, ideally we would still have this data. But our `EventViewModel` is gone and will be replaced by a brand-new instance in the brand-new process. We could store this data in a file or database, but unless the list of Event objects is really long, we could put this data into the saved instance state `Bundle`.

Except for one problem: `Bundle` does not know anything about Event.

In order to be able to put our Event objects into the `Bundle`, we need to make Event be `Parcelable`.

`Parcelable` is a marker interface, reminiscent of Java's `Serializable`, that shows up in many places in the Android SDK. 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. And, of note, `Bundle` supports `Parcelable` objects.

You have two 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. Just do it yourself

Parcelable by Annotation

Annotation processors take `@Things @That @LookLike @These("snippets")` and augment your code. In particular, they can generate code for you to save you typing in a bunch of boilerplate.

Kotlin users have easy access to a `@Parcelize` annotation that can make simple classes — such as Event — be `Parcelable`.

To add `Parcelable` support to a Kotlin class, just:

- Have that class declare that it implements the `Parcelable` interface, though you do not need to override any of its methods, and
- Add the `@Parcelize` annotation to the class:

```
package com.commonware.jetpack.sampler.state

import android.os.Parcelable
import android.os.SystemClock
import kotlinx.android.parcel.Parcelize

@Parcelize
data class Event(
    val message: String,
    val activityHash: Int,
    val viewModelHash: Int,
    val timestamp: Long = SystemClock.elapsedRealtime()
) : Parcelable
```

(from [InstanceState/src/main/java/com/commonware/jetpack/sampler/state/Event.kt](#))

Here we have a revised version of the Event class that adds the @Parcelize annotation and the Parcelable interface. The actual code to support the Parcelable interface will be code-generated, and so we can skip it.

Support for this annotation is supplied by the kotlin-parcelize Gradle plugin, which you will see at the top of many projects, including all of the Kotlin projects in this book:

```
apply plugin: 'kotlin-parcelize'
```

You may see references to a kotlin-android-extensions plugin instead. This used to supply @Parcelize, but that plugin has been deprecated and replaced by kotlin-parcelize.

In addition, there are [many third-party libraries](#) that support Parcelable, including many annotation processors for Java that can accomplish a similar thing to what we get with Kotlin.

Parcelable 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 the Parcelable interface 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`, including:

- 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`
- various specialized methods for particular data types (e.g., `writeSizeF()`) or interfaces (e.g., `writeSerializable()`)

In the case of the generated Event code shown earlier in this chapter, `writeToParcel()` writes out each of our four fields:

```
@Override
public void writeToParcel(Parcel dest, int flags) {
    dest.writeLong(timestamp);
    dest.writeString(message);
    dest.writeInt(activityHash);
    dest.writeInt(viewmodelHash);
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/Event.java](#))

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<Event> CREATOR = new Parcelable.Creator<Event>() {
    @Override
    public Event createFromParcel(Parcel in) {
        return new Event(in);
    }

    @Override
    public Event[] newArray(int size) {
        return new Event[size];
    }
};
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/Event.java](#))

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 Event(Parcel in) {
    timestamp = in.readLong();
    message = in.readString();
    activityHash = in.readInt();
    viewmodelHash = in.readInt();
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/Event.java](#))

You need to read in the *same* values that you wrote out to the `Parcel`, and in the same order.

A State-Aware ViewModel

Version 2.2.0 of the Jetpack lifecycle artifacts added first-class support for tying `ViewModel` and your saved instance state together. The `InstanceState` sample module in the [Sampler](#) and [SamplerJ](#) projects use an `EventViewModel` that still holds our events and start time. However, in this case, we also use the new `ViewModel` capabilities to store that data in the saved instance state `Bundle` and use that data when creating a new `EventViewModel` instance in a new process.

The SavedStateHandle

The first step for making this work is to add a SavedStateHandle constructor parameter to our ViewModel. SavedStateHandle is a wrapper around the saved instance state Bundle that the Jetpack developers introduced.

So, EventViewModel has that constructor parameter:

```
package com.commonware.jetpack.samplerj.state;

import android.os.SystemClock;
import java.util.ArrayList;
import java.util.Random;
import androidx.lifecycle.SavedStateHandle;
import androidx.lifecycle.ViewModel;

public class EventViewModel extends ViewModel {
    private static final String STATE_EVENTS = "events";
    private static final String STATE_START_TIME = "startTime";

    final ArrayList<Event> events;
    final Long startTime;
    private final int id = new Random().nextInt();
    private final SavedStateHandle state;

    public EventViewModel(SavedStateHandle state) {
        this.state = state;

        ArrayList<Event> events = state.get(STATE_EVENTS);

        if (events == null) {
            this.events = new ArrayList<>();
        }
        else {
            this.events = events;
        }

        Long startTime = state.get(STATE_START_TIME);

        if (startTime == null) {
            this.startTime = SystemClock.elapsedRealtime();
            state.set(STATE_START_TIME, this.startTime);
        }
        else {
            this.startTime = startTime;
        }
    }
}
```

UNDERSTANDING PROCESSES

```
}

void addEvent(String message, int activityHash) {
    events.add(new Event(message, activityHash, id));
    state.set(STATE_EVENTS, events);
}

@Override
protected void onCleared() {
    events.clear();
}
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/EventViewModel.java](#))

```
package com.commonsware.jetpack.sampler.state

import android.os.SystemClock
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import java.util.*

private const val STATE_EVENTS = "events"
private const val STATE_START_TIME = "startTime"

class EventViewModel(private val state: SavedStateHandle) : ViewModel() {
    val events: ArrayList<Event> = state.get(STATE_EVENTS) ?: arrayListOf()
    val startTime: Long = state.get(STATE_START_TIME)
        ?: SystemClock.elapsedRealtime().also { state.set(STATE_START_TIME, it) }
    private val id = Random().nextInt()

    fun addEvent(message: String, activityId: Int) {
        events.add(Event(message, activityId, id))
        state.set(STATE_EVENTS, events)
    }

    override fun onCleared() {
        events.clear()
    }
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/sampler/state/EventViewModel.kt](#))

SavedStateHandle uses basic `get()` and `set()` methods for manipulated the saved instance state. Both operate using strings as keys, as with a Bundle. So, in EventViewModel, we initialize our events and startTime properties to be based on the supplied SavedStateHandle. But, if our handle does not contain that data, we

initialize it to be an empty list of events and the current time, respectively.

Whenever we change these properties, we also update the `SavedStateHandle`. Since the value of `startTime` is only defined once, if we need to use the current time for its value, we also use `set()` to save that in the handle. And, in the `addEvent()` function, we not only update the contents of the events `ArrayList`, but we make sure that the `SavedStateHandle` has the latest copy of those events.

Note that `id` is *not* part of the saved instance state. That value will get regenerated for each new `EventViewModel`.

The Results

We do not need to do anything special when retrieving our viewmodels — the Jetpack viewmodel system already knows how to handle `SavedStateHandle` scenarios by default. As a result, our activities look and work much the same as the earlier `LifecycleList` counterparts:

```
package com.commonware.jetpack.samplerj.state;

import android.os.Bundle;
import com.commonware.jetpack.samplerj.state.databinding.ActivityMainBinding;
import java.util.ArrayList;
import java.util.Random;
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProvider;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;

public class MainActivity extends AppCompatActivity {
    private EventAdapter adapter;
    private EventViewModel vm;
    private final int id = new Random().nextInt();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final ActivityMainBinding binding =
            ActivityMainBinding.inflate(getLayoutInflater());

        setContentView(binding.getRoot());

        vm = new ViewModelProvider(this).get(EventViewModel.class);
        adapter = new EventAdapter(getLayoutInflater(), vm.startTime);
```

```
addEvent("onCreate()");

binding.items.setLayoutManager(new LinearLayoutManager(this));
binding.items.addItemDecoration(
    new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
binding.items.setAdapter(adapter);
}

@Override
protected void onStart() {
    super.onStart();

    addEvent("onStart()");
}

@Override
protected void onResume() {
    super.onResume();

    addEvent("onResume()");
}

@Override
protected void onPause() {
    addEvent("onPause()");

    super.onPause();
}

@Override
protected void onStop() {
    addEvent("onStop()");

    super.onStop();
}

@Override
protected void onDestroy() {
    addEvent("onDestroy()");

    super.onDestroy();
}

private void addEvent(String message) {
    vm.addEvent(message, id);
    adapter.submitList(new ArrayList<>(vm.events));
}
}
```

UNDERSTANDING PROCESSES

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/MainActivity.java](#))

```
package com.commonsware.jetpack.sampler.state

import android.os.Bundle
import androidx.activity.viewModels
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.jetpack.sampler.state.databinding.ActivityMainBinding
import java.util.*
import kotlin.collections.ArrayList

class MainActivity : AppCompatActivity() {
    private val vm: EventViewModel by viewModels()

    private lateinit var adapter: EventAdapter
    private val id = Random().nextInt()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)

        adapter = EventAdapter(layoutInflater, vm.startTime)
        addEvent("onCreate()")

        binding.items.layoutManager = LinearLayoutManager(this)
        binding.items.addItemDecoration(
            DividerItemDecoration(this, DividerItemDecoration.VERTICAL)
        )
        binding.items.adapter = adapter
    }

    override fun onStart() {
        super.onStart()

        addEvent("onStart()")
    }

    override fun onResume() {
        super.onResume()

        addEvent("onResume()")
    }
}
```

```
override fun onPause() {
    addEvent("onPause()")

    super.onPause()
}

override fun onStop() {
    addEvent("onStop()")

    super.onStop()
}

override fun onDestroy() {
    addEvent("onDestroy()")

    super.onDestroy()
}

private fun addEvent(message: String) {
    vm.addEvent(message, id)
    adapter.submitList(ArrayList(vm.events))
}
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/sampler/state/MainActivity.kt](#))

Seeing the effect of our saved instance state support is tricky. We need to have our process be terminated while leaving the task alone. That means we cannot swipe our task off of the overview screen, as that terminates the task. Similarly, the stop-process buttons in Android Studio toolbars (with a red square icon) seem to terminate the task as well.

One way to see this work is to use the command line.

The Android SDK ships with an adb command-line utility. We can use this to communicate with a running emulator or debuggable device. You will find adb in the platform-tools/ directory of your Android SDK installation.

In particular:

- adb shell says “give me access to a Linux-style shell inside of Android”
- adb shell am says “run the am command in that shell”, where am is the “activity manager” tool
- adb shell am kill ... says “kill the process identified by the supplied application ID” (shown here as ...)

UNDERSTANDING PROCESSES

To experiment with instance states:

- Run the sample app
- Undergo configuration changes, if desired
- Press HOME, to move the app to the background and trigger an `onSaveInstanceState()` call
- Run `adb shell am kill com.commonware.jetpack.sampler.state` (for the Kotlin edition) or `adb shell am kill com.commonware.jetpack.samplerj.state` (for the Java edition) to terminate the background process
- Bring up the overview screen and tap on the entry for this sample app

In the original `LifecycleList` sample, configuration changes would show new instance ID codes for the activity, but the `EventViewModel` instance ID code would be the same, as we reuse the existing viewmodel on a configuration change. Now, though, we are terminating the process, so we should see a new viewmodel ID code for the newer events:



Figure 155: InstanceState Sample App, Following Above Script

However, we still have the original events and the original start time — we did not start over with an empty `EventViewModel`, because we populated it from the saved

instance state Bundle.

Binding Your Data

So far, to update our UI, we have been pushing data into widgets from our Java or Kotlin code. For example, we have been updating the text property of a `TextView`.

Data binding, in general, refers to frameworks or libraries designed to *pull* data into the UI. UI definitions — such as Android layout resources — contain information about how to populate widgets from supplied model objects.

This chapter explores Android’s data binding support and how to use it to perhaps simplify your Android app development. And, we will also see why the word “perhaps” is in that previous sentence.

The Basic Steps

The `DataBind` sample module in the [Sampler](#) and [SamplerJ](#) projects adds to the `InstanceState` sample from the preceding chapter. The primary difference is that we will fill in the `RecyclerView` rows using data binding.

Enabling Data Binding

Data binding is an opt-in feature, in part because it can slow down the build process. For small projects like the ones in this book, you will not notice the overhead of the data binding tools, but that overhead becomes more annoying as projects get larger.

To opt into data binding, we need to enable it, by adding another closure to our module’s `build.gradle` file’s `android` closure:

```
buildFeatures {  
    dataBinding = true  
    viewBinding true  
}
```

(from [DataBind/build.gradle](#))

This works similar to how we enabled view binding [earlier in the book](#). In this case, we are enabling both view binding and data binding.

Also, Kotlin projects using data binding should add the `kotlin-kapt` plugin:

```
apply plugin: 'kotlin-kapt'
```

(from [DataBind/build.gradle](#))

This plugin enables annotation processing for Kotlin source files. Using annotations is not absolutely required in a data binding project, but it is somewhat common, and this plugin is needed so the data binding code can process those annotations.

Augmenting the Layout

The real fun begins with the layout for our RecyclerView row. The original edition of this layout resource was a typical ConstraintLayout with our TextView widgets:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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="wrap_content"  
    android:padding="@dimen/content_padding">  
  
    <TextView  
        android:id="@+id/activityHash"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintTop_toTopOf="parent"  
        tools:text="0x12345678" />  
  
    <TextView  
        android:id="@+id/viewmodelHash"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"
```

BINDING YOUR DATA

```
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintTop_toBottomOf="@id/activityHash"
tools:text="0x90ABCDEF" />

<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="4dp"
    android:layout_marginStart="4dp"
    app:barrierDirection="start"
    app:constraint_referenced_ids="activityHash,viewModelHash" />

<TextView
    android:id="@+id/timestamp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="01:23" />

<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@id/barrier"
    app:layout_constraintStart_toEndOf="@id/timestamp"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [InstanceState/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>

    <data>

        <import type="android.text.format.DateUtils" />

        <variable
```

```
        name="event"
        type="com.commonware.jetpack.sampler.databind.Event" />

<variable
    name="elapsedSeconds"
    type="Long" />
</data>

<androidx.constraintlayout.widget.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="wrap_content"
    android:padding="@dimen/content_padding">

    <TextView
        android:id="@+id/activityHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{Integer.toHexString(event.activityHash)}"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="0x12345678" />

    <TextView
        android:id="@+id/viewmodelHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{Integer.toHexString(event.viewmodelHash)}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@id/activityHash"
        tools:text="0x90ABCDEF" />

    <androidx.constraintlayout.widget.Barrier
        android:id="@+id/barrier"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="4dp"
        android:layout_marginStart="4dp"
        app:barrierDirection="start"
        app:constraint_referenced_ids="activityHash,viewModelHash" />

    <TextView
        android:id="@+id/timestamp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

BINDING YOUR DATA

```
        android:text="@{DateUtils.formatElapsedTime(elapsedSeconds)}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="01:23" />

<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{event.message}"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@id/barrier"
    app:layout_constraintStart_toEndOf="@id/timestamp"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [DataBind/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. Only layout resources with a root `<layout>` element are processed by the data binding portion of the build system.

That `<layout>` element then has two children. The second child is our `ConstraintLayout`, 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.

Inside of `<data>` we have three elements.

Two are `<variable>` elements. These identify and describe the objects that we can pull data from. In our case, we have an event variable that is an instance of our Event model class, and we have an `elapsedSeconds` value that represents the number of seconds that have elapsed between the start of the app and this event. As we will see, our Java or Kotlin code will now “bind” those objects into our layout, rather than update widgets directly.

The other element is `<import>`. This works like `import` statements in Java or Kotlin, indicating a particular class that we would like to reference. In the world of data binding, mostly we use `<import>` for classes where we wish to refer to static

methods or fields (and their Kotlin equivalents).

Our four `TextView` widgets now have `android:text` attributes, where they had none before. That is because the earlier version of this sample relied upon Java or Kotlin code to push data into the widgets, and now we are going to pull data in using data binding.

Those `android:text` attributes use “binding expressions”. A binding expression is identified in a data binding-enhanced layout through `@{}` syntax, with the actual expression between the braces. Those expressions use a language syntax that looks a lot like Java or Kotlin expressions, and they can reference:

- Our `<variable>` values
- Global functions and properties on the classes listed in `<import>` elements
- Global functions and properties on any `java.lang` classes
- A few other magic values, such as `context` to get a `Context`, should we need one
- Properties of other widgets in this same layout resource

The `activityHash TextView` has `android:text="@{Integer.toHexString(event.activityHash)}"`. The binding expression works like its Java or Kotlin counterparts, taking our `activityHash` value out of our event and formatting it as a hex string. The text of this `TextView` will then contain that hex string.

The `android:text` attributes of the other `TextView` widgets work the same:

- `viewModelHash` uses `Integer.toHexString(event.viewModelHash)` to fill in the text with the hex string of that hash code
- `timestamp` uses `DateUtils.formatElapsedTime(elapsedSeconds)` to format the `elapsedSeconds` value as an elapsed time
- `message` just uses `event.message` to bind from the message from the `Event`

While we happen to use `android:text` for all four widgets, binding expressions can be applied to just about any attribute, such as `android:checked` for the checked state of a `CompoundButton`.

Updating the Model

Data binding has its limits. One limit is that it can only access public functions and properties. So, we need to ensure that everything we need is now public in our

Event class, both in Java:

```
package com.commonware.jetpack.samplerj.databind;

import android.os.Parcel;
import android.os.Parcelable;
import android.os.SystemClock;

public class Event implements Parcelable {
    public final long timestamp;
    public final String message;
    public final int activityHash;
    public final int viewmodelHash;

    Event(String message, int activityHash, int viewmodelHash) {
        this.message = message;
        this.activityHash = activityHash;
        this.viewmodelHash = viewmodelHash;
        this.timestamp = SystemClock.elapsedRealtime();
    }

    protected Event(Parcel in) {
        timestamp = in.readLong();
        message = in.readString();
        activityHash = in.readInt();
        viewmodelHash = in.readInt();
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeLong(timestamp);
        dest.writeString(message);
        dest.writeInt(activityHash);
        dest.writeInt(viewmodelHash);
    }

    @SuppressWarnings("unused")
    public static final Parcelable.Creator<Event> CREATOR = new Parcelable.Creator<Event>() {
        @Override
        public Event createFromParcel(Parcel in) {
            return new Event(in);
        }

        @Override
        public Event[] newArray(int size) {
            return new Event[size];
        }
    };
}
```

(from [DataBind/src/main/java/com/commonware/jetpack/samplerj/databind/Event.java](#))

...and Kotlin:


```
package com.commonware.jetpack.sampler.databind

import android.os.Parcelable
import android.os.SystemClock
import kotlinx.android.parcel.Parcelize

@Parcelize
data class Event(
    val message: String,
    val activityHash: Int,
    val viewModelHash: Int,
    val timestamp: Long = SystemClock.elapsedRealtime()
) : Parcelable
```

(from [DataBind/src/main/java/com/commonware/jetpack/sampler/databind/Event.kt](#))

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 Event object for this layout resource.

That is handled via some modifications to our EventAdapter and EventViewHolder, to get at a “binding object” for an inflated layout and then call functions on it to supply our variables.

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. As with view binding, 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.commonware.jetpack.sampler.databind.databinding.RowBinding;
```

(normally, projects would not have databind in their own application IDs, so the near-duplication that you see here is peculiar to this project)

This is a subclass of ViewDataBinding, supplied by the androidx.databinding

BINDING YOUR DATA

libraries that are 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 — just like we have used with view binding

Our revised version of `EventAdapter` uses 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. We use this in `onCreateViewHolder()` and pass the `RowBinding` into our `EventViewHolder`:

```
@NonNull
@Override
public EventViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                         int viewType) {
    RowBinding binding = RowBinding.inflate(inflater, parent, false);

    return new EventViewHolder(binding, startTime);
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/samplerj/databind/EventAdapter.java](#))

```
override fun onCreateViewHolder(
    parent: ViewGroup,
    viewType: Int
): EventViewHolder {
    val binding = RowBinding.inflate(inflater, parent, false)

    return EventViewHolder(binding, startTime)
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/sampler/databind/EventAdapter.kt](#))

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 event variable becomes `setEvent()` and our `elapsedSeconds` variable becomes `setElapsedSeconds()`. When we call `setEvent()` and `setElapsedSeconds()`, the generated code will use those objects to populate our `TextView` widgets, applying the binding expression from our `android:text` attributes.

To accomplish this, our revised `EventViewHolder` holds onto the `RowBinding` and uses it in `bindTo()`:

```
package com.commonware.jetpack.samplerj.databind;

import com.commonware.jetpack.samplerj.databind.databinding.RowBinding;
import androidx.recyclerview.widget.RecyclerView;

class EventViewHolder extends RecyclerView.ViewHolder {
    private final long startTime;
    private final RowBinding row;

    EventViewHolder(RowBinding row, long startTime) {
        super(row.getRoot());

        this.row = row;
        this.startTime = startTime;
    }

    void bindTo(Event event) {
        row.setElapsedSeconds((event.timestamp - startTime)/1000);
        row.setEvent(event);
        row.executePendingBindings();
    }
}
```

(from `DataBind/src/main/java/com/commonware/jetpack/samplerj/databind/EventViewHolder.java`)

```
package com.commonware.jetpack.sampler.databind

import androidx.recyclerview.widget.RecyclerView
import com.commonware.jetpack.sampler.databind.databinding.RowBinding

class EventViewHolder(val row: RowBinding, private val startTime: Long) :
    RecyclerView.ViewHolder(row.root) {
    fun bindTo(event: Event) {
```

BINDING YOUR DATA

```
row.elapsedSeconds = (event.timestamp - startTime) / 1000
row.event = event
row.executePendingBindings()
}
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/sampler/databind/EventViewHolder.kt](#))

We also call `executePendingBindings()` on the `RowBinding`. This is needed when populating items in a `RecyclerView`, to ensure that those binding expressions are evaluated and applied immediately rather than a bit later. In cases not involving `RecyclerView`, though, it is usually safe to skip the `executePendingBindings()` call.

Getting the Root View

When we chain to the superclass constructor in a `RecyclerView.ViewHolder`, we need to pass the `View` that is the UI for that particular bit of UI that the `ViewHolder` is managing. To get the root view of a layout associated with a binding object, call `getRoot()` on the binding object, as we do with view binding. That's what `EventViewHolder` does, passing the `getRoot()` results to the `RecyclerView.ViewHolder` constructor.

Results

Visually, this app is the same as before. 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()` and `setText()` a few times.

Why Bother?

Some Android experts love data binding. Others hate it. Compared to a lot of things in Android app development, opinions of data binding are wide and varied.

Developers that love data binding seem to focus a lot on the “separation of concerns” that data binding helps to enforce. Your Java/Kotlin code can stop thinking about widget details quite so much, with a lot of that code moving to the layout resources.

Detractors point out that:

- Data binding slows down the build process, as there is more code generation

- that needs to go on, far more than simple view binding
- Data binding can be difficult to debug, as more code is hidden from view, buried in data binding expressions and generated classes
- Data binding adds a bit more bloat to a project, adding three libraries and these generated classes, where the user gains very little from the results — view binding is smaller

Google promotes data binding as part of the Jetpack. However, whereas some aspects of Jetpack are nearly unavoidable, data binding is optional. You can use it if you like or skip it if you like.

The Other Common “Gimme the Views” Options

We have seen both data binding and view binding as ways of interacting with the widgets in our layout resources.

Believe it or not... there are others, though in general they are no longer recommended.

`findViewById()`

The original solution, dating back to Android 1.0, is a method called `findViewById()`. This method is available on `Activity`, `View`, and `ViewGroup`. You pass in a widget resource ID (`R.id.whatever`) and it returns the first `View` that it finds that matches that ID... or null if it fails to find something:

```
val button = findViewById<Button>(R.id.whatever)
```

The problem is that there is no validation on the widget ID that you supply. So, while we like to pretend that `findViewById()` will always return our desired widget, that is based on some assumptions:

- The widget ID is valid for this context — for example, you did not provide a widget ID from Layout A to a `findViewById()` call that is tied to something that instead inflated Layout B
- The layout is already inflated by this point — a mistake many early Android developers made was to call `findViewById()` before `setContentView()`, for example

The compiler will let you call `findViewById()` on any `View` or `ViewGroup` passing in

any `R.id` resource... even if there is no possible way that a widget with that ID will be found. Most of the time, you will know what widget IDs to use in what circumstances and will create valid `findViewById()` calls. Sometimes, though, you will make mistakes... and then it comes down to testing as to whether or not you catch those mistakes before they start affecting users. View binding helps to eliminate that risk.

As a result, `findViewById()` is available but is no longer recommended for most developers. What view binding and data binding do is give us code-generated classes that limit our `findViewById()` usage to cases that are far more likely to succeed.

Kotlin Synthetic Accessors

Kotlin synthetic accessors come “for free” simply by adding the `kotlin-android-extensions` plugin to a module.

The effect is that we can add an import statement that references our layout resource (`import kotlinx.android.synthetic.main.activity_main.*`, referencing the `activity_main` resource). With that, we get properties for each of the named widgets in the layout. Under the covers, generated code will perform the `findViewById()` call for us.

The larger the project, though, the greater the risk is that you will use the wrong imports. Once you start getting dozens or hundreds of layout resources, making sure that you use the right imports starts to become a challenge. The compiler will be very happy to let you use any import you want. However, if you try referencing a widget from Layout X and you are really using Layout Y, it is likely that you will wind up crashing with a `NullPointerException` somewhere along the line.

Also, this approach is only available for Kotlin code — it adds no value to a Java project.

Most importantly, though, this plugin is deprecated by JetBrains as of Kotlin 1.4.20. So, while Kotlin synthetic accessors were “the go-to” Kotlin solution as recently as 2018, they are no longer recommended, even by their developers.

Defining and Using Styles

Android offers styles and themes, filling the same sort of role that CSS does in Web development. We have seen a little bit about themes, such as having a custom theme that defines the core colors to be used by your app.

On the whole, styles and themes are powerful yet confusing tools in the Android developer's toolbox. The “confusing” aspect stems from documentation that has ranged from “simply not existing” to “limited and mystifying” over the years. As such, most Android app development seems to avoid styles and themes to the extent possible, settling instead for directly configuring widgets in layout files. This works, and for smaller projects it is perfectly reasonable. The larger the project, the greater the likelihood is that you will benefit from using styles and themes, just as a larger Web app is more likely than a smaller one to benefit from a well-designed set of CSS stylesheets.

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. 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.

The CustomStyle sample module in the [Sampler](#) and [SamplerJ](#) projects is based off of the previous InstanceState sample, where we are showing a list of lifecycle events. In this edition of the sample, we will define and apply a style resource to

some of the `TextView` widgets in the list rows.

Styles and themes are defined using a style resource. By convention, style resources go in `res/values/styles.xml`, though the actual filename does not matter, so long as it is in a values resource directory.

Our `res/values/styles.xml` defines two style resources. One, `AppTheme`, is our custom application theme, and we will revisit it [later in the chapter](#).

The other is `CustomText`:

```
<style name="CustomText">
  <item name="android:textColor">?colorAccent</item>
  <item name="android:textStyle">italic</item>
  <item name="android:fontFamily">monospace</item>
</style>
```

(from [CustomStyle/src/main/res/values/styles.xml](#))

In the next sections, we will explore more about what this style does and how we are using it.

Elements of Style

There are a few elements to consider when working with style resources:

- Where do you put the style attributes to say you want to apply a style?
- What attributes can you define via a style?
- What values can those attributes have in a style definition?

The Locations Where We Use Styles

To indicate that we want to use a `<style>` to style a widget, we apply the `style` attribute to the widget, with a reference to our style resource (e.g., `@style/CustomText`). So, for example, our `res/layout/row.xml` resource uses `style="@style/CustomText"` on the `activityHash` and `viewModelHash` widgets, but not the others:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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="wrap_content"
android:padding="@dimen/content_padding">

<TextView
    android:id="@+id/activityHash"
    style="@style/CustomText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="0x12345678" />

<TextView
    android:id="@+id/viewmodelHash"
    style="@style/CustomText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/activityHash"
    tools:text="0x90ABCDEF" />

<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="4dp"
    android:layout_marginStart="4dp"
    app:barrierDirection="start"
    app:constraint_referenced_ids="activityHash,viewModelHash" />

<TextView
    android:id="@+id/timestamp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="01:23" />

<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
```

```
app:layout_constraintEnd_toStartOf="@id/barrier"
app:layout_constraintStart_toEndOf="@id/timestamp"
app:layout_constraintTop_toTopOf="parent"
tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [CustomStyle/src/main/res/layout/row.xml](#))

If you apply the style attribute to a widget, it affects only that widget.

The style attribute can be applied to a container, to affect that container. However, doing this does *not* automatically style widgets that reside inside of the container.

You can also apply a style to an activity or an application as a whole... in the form of a theme. We will explain the differences more [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, or it shows up in the Attributes pane of the Android Studio graphical layout designer, you can put it in a style.

If we go back to CustomText, we will see that our style has three `<item>` elements, identifying three attributes that we wish to control:

```
<style name="CustomText">
  <item name="android:textColor">?colorAccent</item>
  <item name="android:textStyle">italic</item>
  <item name="android:fontFamily">monospace</item>
</style>
```

(from [CustomStyle/src/main/res/values/styles.xml](#))

Note that Android will ignore invalid styles. If you put an attribute in a style that is unused by something that you apply the style to, the attribute is ignored. It does not crash the app. It so happens that TextView has `android:textColor`, `android:textStyle`, and `android:fontFamily` attributes, to control the text color, style (bold, italic, etc.) and font (e.g., monospace) respectively. If we applied this style to the ConstraintLayout at the root of row.xml, all three style attribute values would be ignored, as a ConstraintLayout does not have a text color, style, or font.

Also, while layout directives, such as `android:layout_width`, can be put in a style,

usually they are not. Styles tend to be limited to other aspects of look-and-feel: colors, fonts, etc.

The Possible Values

Sometimes, the value that you will give those attributes in the style will be some constant, like 30sp or #FFFF0000.

Sometimes, the value will appear to be string, but in reality it is one of a limited number of possible “enumerated” values. For example, `android:textStyle` supports `italic` as a value, as the sample project is using in `CustomText`. It does not support `aeroasdfsdf` as a value.

Sometimes, the value will be a reference to a resource, such as `@dimen/standard_padding` or `@color/really_red`.

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 `?` and `?android:attr/` syntax.

For example, the `android:textColor` attribute in the `CustomText` style does not have a value of `#D81B60` or `@color/colorAccent`, even though both of those are the color that we actually wind up applying. Instead, `android:textColor` has `?colorAccent`. This says “go find the `colorAccent` value defined for our theme and use it here too”. Our theme is `AppTheme`, and it has a `colorAccent` value:

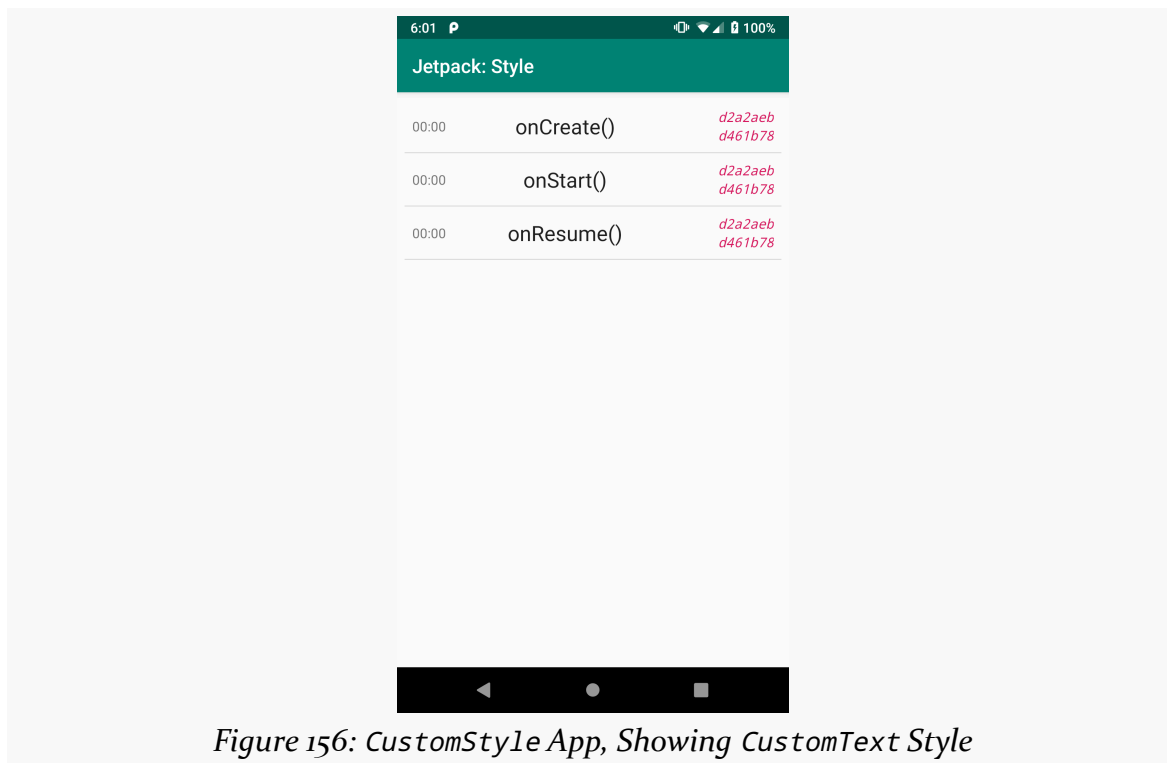
```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
  <!-- Customize your theme here. -->
  <item name="colorPrimary">@color/colorPrimary</item>
  <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
  <item name="colorAccent">@color/colorAccent</item>
</style>
```

(from [CustomStyle/src/main/res/values/styles.xml](#))

This way, if we change the theme’s `colorAccent`, we *also* change the `android:textColor` of `CustomText` to match.

The Results

We applied the CustomText style to two of our four TextView widgets, so those two are now red italic monospace:



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. They affect properties that apply to the entire activity, such as the default colors to use. They also can define default attributes to use for widgets used by that activity, without manually applying style attributes to those widgets.

The CustomTheme sample module in the [Sampler](#) and [SamplerJ](#) projects is a variation of the preceding sample, where we want to make TextView widgets display red italic monospace text. This time, though, we are going to work with our AppTheme style resource as a theme.

The Locations Where We Apply a Theme

A theme usually is applied via an `android:theme` attribute in the manifest. If you apply it to an `<activity>`, that activity will use the designated theme. If you apply it to an `<application>`, that theme will be used for *all* activities in the app... except for any that override it with their own `android:theme` attribute.

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 (e.g., `@style/AppTheme`). This includes themes defined in libraries that you are using (`@style/Theme.AppCompat.Dialog`). 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.Material.Dialog` or `@android:style/Theme.Material.Light`.

So, our sample app uses `android:theme` on the `<application>`, pointing to `AppTheme`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonware.jetpack.sampler.theme"
    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>
```

(from [CustomTheme/src/main/AndroidManifest.xml](#))

The Theme Declaration

Themes are just style resources.

So, our AppTheme appears in `res/values/styles.xml` alongside CustomText, though with slight alterations from what we used in the previous sample:

```
<resources>

  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
    <item name="android:textViewStyle">@style/CustomText</item>
  </style>

  <style name="CustomText" parent="android:Widget.Material.TextView">
    <item name="android:textColor">?colorAccent</item>
    <item name="android:textStyle">italic</item>
    <item name="android:fontFamily">monospace</item>
  </style>

</resources>
```

(from [CustomTheme/src/main/res/values/styles.xml](#))

The Parent, and an AppCompatActivity Recap

Any style resource can have a parent attribute. This is an inheritance model: anything using the style resource gets the attributes defined both directly in that style resource plus any defined by its parent.

A theme always has a parent, and that parent should be another theme. There are hundreds, if not thousands, of attributes to be configured on a theme, so you want to inherit the vast majority of them. AppTheme is declared to have Theme.AppCompat.Light.DarkActionBar as its parent, so the activity in this app will not only get the attributes defined directly in AppTheme but all the attributes defined by Theme.AppCompat.Light.DarkActionBar.

Theme.AppCompat.Light.DarkActionBar is part of the AppCompatActivity system. If you have an activity that extends AppCompatActivity — as all of the ones in this book do — that activity *must* use a theme that inherits from Theme.AppCompat. Theme.AppCompat.Light.DarkActionBar itself inherits from Theme.AppCompat, so AppTheme indirectly inherits from Theme.AppCompat.

The Style Override

A theme can have attributes that override ones defined in parents. `Theme.AppCompat` declares `colorPrimary`, `colorPrimaryDark`, and `colorAccent` attributes — `AppTheme` overrides them and provides the values that should be used in the app.

One particular class of attributes defined in parents are references to style resources that should be used to style particular types of widgets. For example, `android:textViewStyle` says “this is the style resource that should be used for all `TextView` widgets”. `AppTheme` has this attribute and points it to `CustomText`, so now the `CustomText` defines the default look for all `TextView` widgets used in activities that use `AppTheme`.

However, we now have to consider the parent of a regular (non-theme) style. When we use a style attribute to apply a style resource to a widget, the attributes defined in the style are used to override those from the default style for the type of widget, such as `TextView`. The `CustomStyle` sample module did not have a parent for `CustomText`, and the three attributes defined by `CustomText` would override whatever the defaults are for `TextView`. However, in the `CustomTheme` project, `CustomText` is the default, courtesy of it being applied via `android:textViewStyle`. As a result, we need `CustomText` to have values for all relevant attributes, mostly through inheritance from some parent. There are three major possibilities for the parent value for an `AppCompat` project like this one:

- If there is an `AppCompat` style for that widget (e.g., `Widget.AppCompat.Button`), use it as the parent
- If there is no `AppCompat` style, and the project has a `minSdkVersion` of 21 or higher, look for a `Material` style for that widget, such as the `android:Widget.Material.TextView` parent used here (the `android:` prefix is because `Widget.Material.TextView` comes from the framework, while `Widget.AppCompat.Button` or `Theme.AppCompat.Light.DarkActionBar` comes from a library)
- If there is no `AppCompat` style, and the project has a lower `minSdkVersion` than 21... the story gets *very* complicated and is well outside the scope of this book

The Result

Since our `CustomText` is now being applied by default for all `TextView` widgets, our row can go back to having no style attributes:


```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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="wrap_content"
    android:padding="@dimen/content_padding">

    <TextView
        android:id="@+id/activityHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="0x12345678" />

    <TextView
        android:id="@+id/viewmodelHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@id/activityHash"
        tools:text="0x90ABCDEF" />

    <androidx.constraintlayout.widget.Barrier
        android:id="@+id/barrier"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="4dp"
        android:layout_marginStart="4dp"
        app:barrierDirection="start"
        app:constraint_referenced_ids="activityHash,viewModelHash" />

    <TextView
        android:id="@+id/timestamp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="01:23" />

    <TextView
        android:id="@+id/message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

DEFINING AND USING STYLES

```
android:textAppearance="?android:attr/textAppearanceLarge"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toStartOf="@id/barrier"
app:layout_constraintStart_toEndOf="@id/timestamp"
app:layout_constraintTop_toTopOf="parent"
tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [CustomTheme/src/main/res/layout/row.xml](#))

Moreover, our red italic monospace look will be applied to all four TextView widgets in the row, not just the two that we manually specified in the CustomStyle sample:

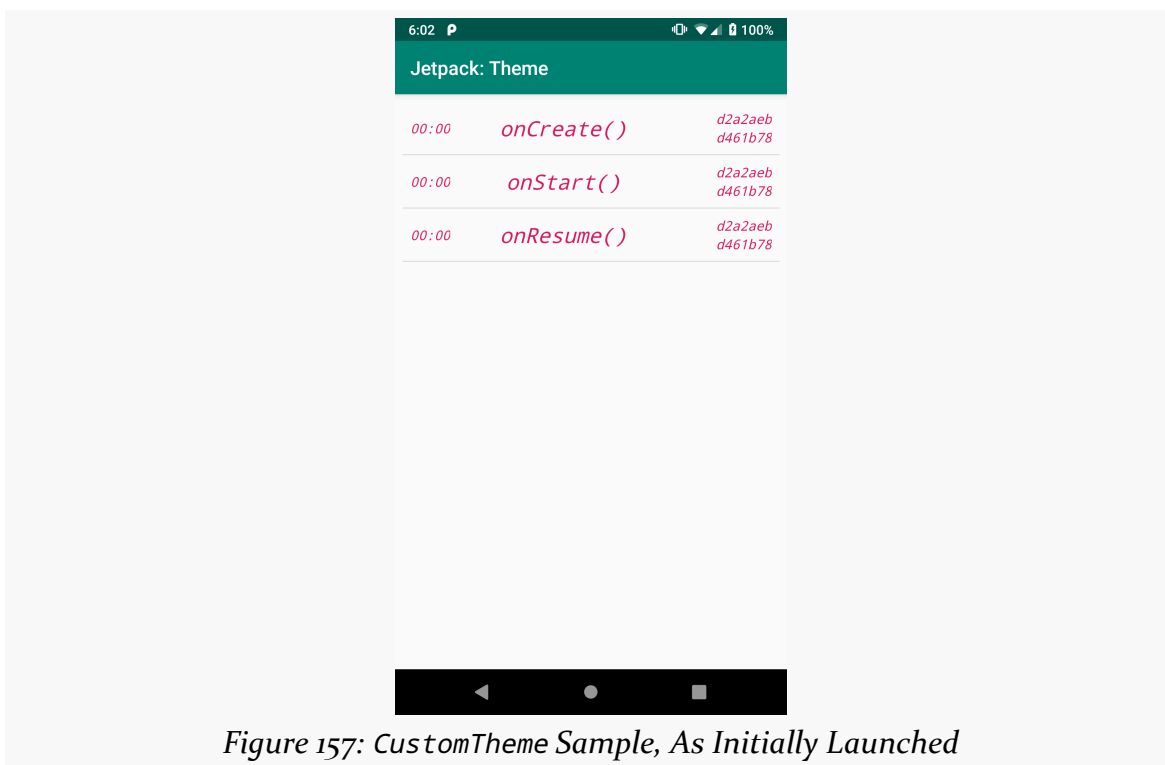


Figure 157: CustomTheme Sample, As Initially Launched

Android 10 Dark Mode

Android 10 offers a system-level option to enable “dark mode”. In dark mode, light UI backgrounds get flipped to dark ones. This primarily affects system UI, but apps can elect to react to this change as well, or otherwise support a dark theme for their apps.

Partly, this is for the user experience. People using their devices at night can do so more easily if the UI is darker and therefore offers less glare. This is why navigation apps often switch into a dark mode at different points (e.g., when ambient light seems to be low), so drivers do not have this bright light shining at them constantly. Also, some users may have visual impairments or other conditions where such glare is a bigger problem than for other people. Plus, with some types of modern displays, black pixels consume less power.

Users can switch to dark mode via the Settings app and the “Dark theme” option in the Display screen:

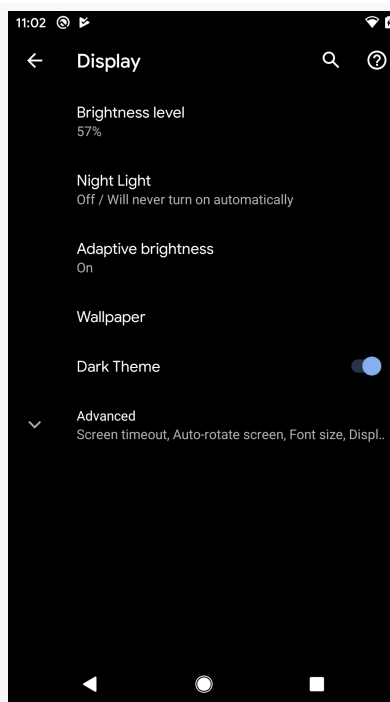


Figure 158: Dark Theme in Settings

The user can also add a tile to the notification shade to be able to rapidly toggle between normal and dark modes:

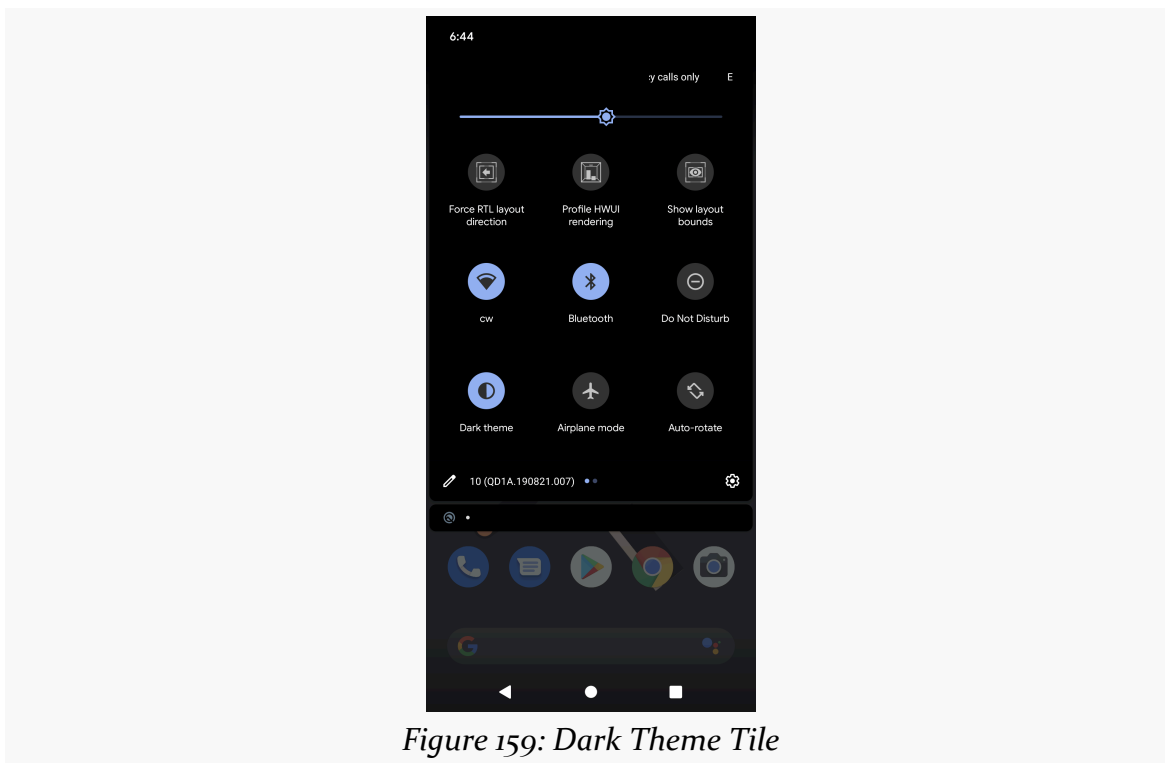


Figure 159: Dark Theme Tile

There are three main ways of handling dark mode in your app, besides ignoring it entirely.

The Dark-All-The-Time Solution

The simplest solution for supporting dark mode is simply to always have a dark theme. This means you have just one theme with one set of colors and artwork, to minimize the work of graphic designers. The user gets the benefits all the time, and the dark theme benefits users across Android versions (not just Android 10 users).

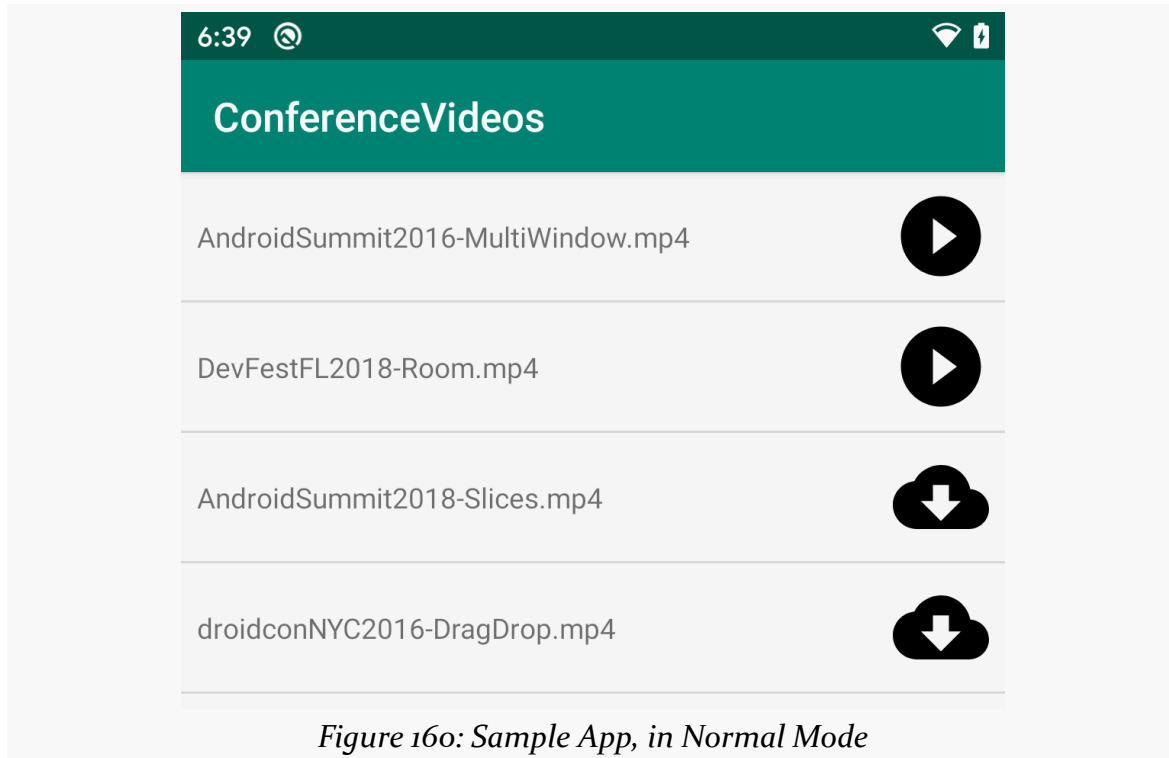
The System Override Solution

You could try to cheat a bit and have the system create a dark theme for you on the fly. For that, add this entry to the `<style>` element for your custom theme:

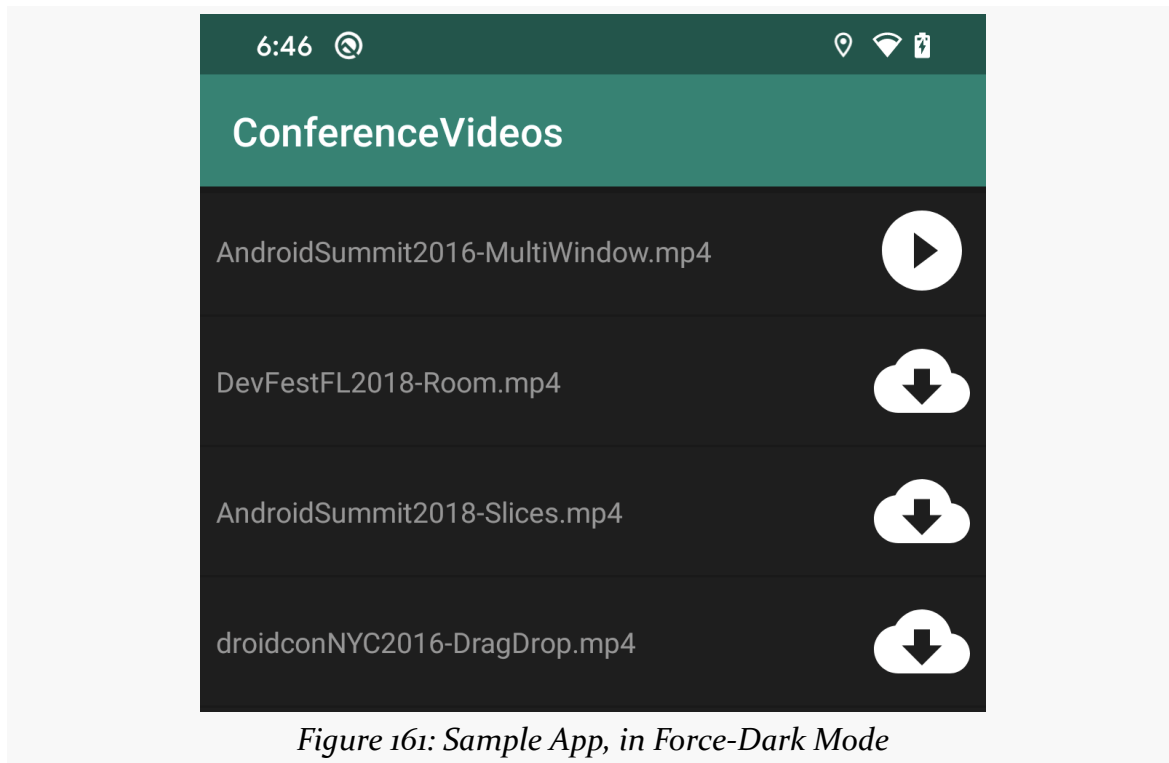
```
<item name="android:forceDarkAllowed">true</item>
```

Then, on Android 10 and higher devices, the system will examine your UI and swap colors to try to make the app appear dark. It even has the smarts to determine whether an `ImageView` appears to be containing an icon (that might be converted) or a photo (that should not be converted).

So, in the default mode, you might have:



...while if the user opts into the dark mode, `android:forceDarkAllowed="true"` will give the user:



This is quick and easy. However:

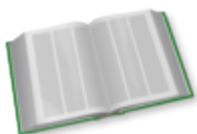
- You do not have any control over the color substitutions, which may make your designers unhappy
- Some things may get converted by accident, requiring you to add `android:forceDarkAllowed="false"` to individual widgets to get them to be left alone
- This only works on Android 10 and higher, so you will have different behavior by OS version

The DayNight Solution

Google's preferred solution is for you to use a theme that adapts based upon whether the device is in dark mode or not. That way, you can have a light theme "normally" while having a dark theme in dark mode.

In particular, AppCompatActivity supports this via its DayNight theme family. The basic recipe is:

- Change your theme's parent theme to `Theme.AppCompat.DayNight` (or to some other theme that extends `Theme.AppCompat.DayNight`).
- Define alternative colors, icons, and other resources in `-night` resource sets. For example, your regular ("day") colors might be in `res/values/colors.xml`, while the "night" colors might be in `res/values-night/colors.xml`. If you use the same name for the individual resources (e.g., `primary`), Android will choose the proper value to use depending on whether dark mode is enabled.



You can learn more about the DayNight option in the "Dark Mode" chapter of [Elements of Android Q!](#)

The Material Components for Android

The example shown in this chapter uses a theme based on `Theme.AppCompat`. All of the sample apps in the `Sampler` and `SamplerJ` projects use themes based on `Theme.AppCompat`.

However, `HelloWorld` and `HelloWorldJ` do not. As we saw [earlier in the book](#), those samples use a theme based on something called `Theme.MaterialComponents`.

This comes from a library known as [the Material Components for Android](#). Principally, this library offers a series of widgets that implement some of [the UI components seen in Material Design](#), from bottom navigation bars to snackbars to "FABs" (floating action buttons). This library also uses a different theme system that uses a different set of color names.

In the author's opinion, the Material Components for Android represents a fine library for experienced Android developers, but [it is a poor choice for newcomers](#). That is why the book continues to use `Theme.AppCompat` — the results that you get are more in line with what documentation and other existing written material covers.

Note, though, that the Android Studio new-project wizard will create a project that uses the Material Components for Android. To switch to AppCompatActivity:

- Replace the theme and its colors with ones from an AppCompatActivity-based project, such as one from this book
- Remove `com.google.android.material:material` from your module's list of dependencies (as you will no longer be needing it)

Context **Anti-Pattern: Using Application Everywhere**

We covered the Application implementation of Context [earlier in the book](#). There, we saw that an Application is a process-wide singleton, so we cannot somehow leak it by holding some reference to it. In effect, it is pre-leaked for us.

Some might wonder why we would ever need any other type of Context. If we need Context for so many things, and we have this one Context that is available all the time... why not use it for everything?

The details are a bit complicated. The fundamental rule is fairly simple, though: never use Application for anything involving the UI, as it may not apply styles and themes correctly.

When it comes time to inflate layouts, we need a LayoutInflater that is aware of the styles and themes to use — otherwise, whatever we requested in those styles and themes will be ignored. But the theme might be specified on a per-activity basis. Right now, we have been looking at apps with a single activity, but it is possible — perhaps even likely — that your app will have [more than one activity](#). Those activities might have different themes. If we get our LayoutInflater from an Activity, it will take the theme into account. If we get our LayoutInflater from the Application, it will not, as the Application has no idea where and how the LayoutInflater will be used.

Using Application for non-UI concerns, such as for working with [files](#) or [databases](#), is reasonable, as there Application is as good as Activity or any other kind of Context. You do not have to go out of your way to use Application in general, but if you have your own singletons that need a Context, Applications will help to avoid memory leaks.

So, use Application:

- If you need a long-lived Context, and
- That Context is not going to be used for setting up your UI

Configuring the App Bar

Each of our apps' activities has had this green bar across the top:

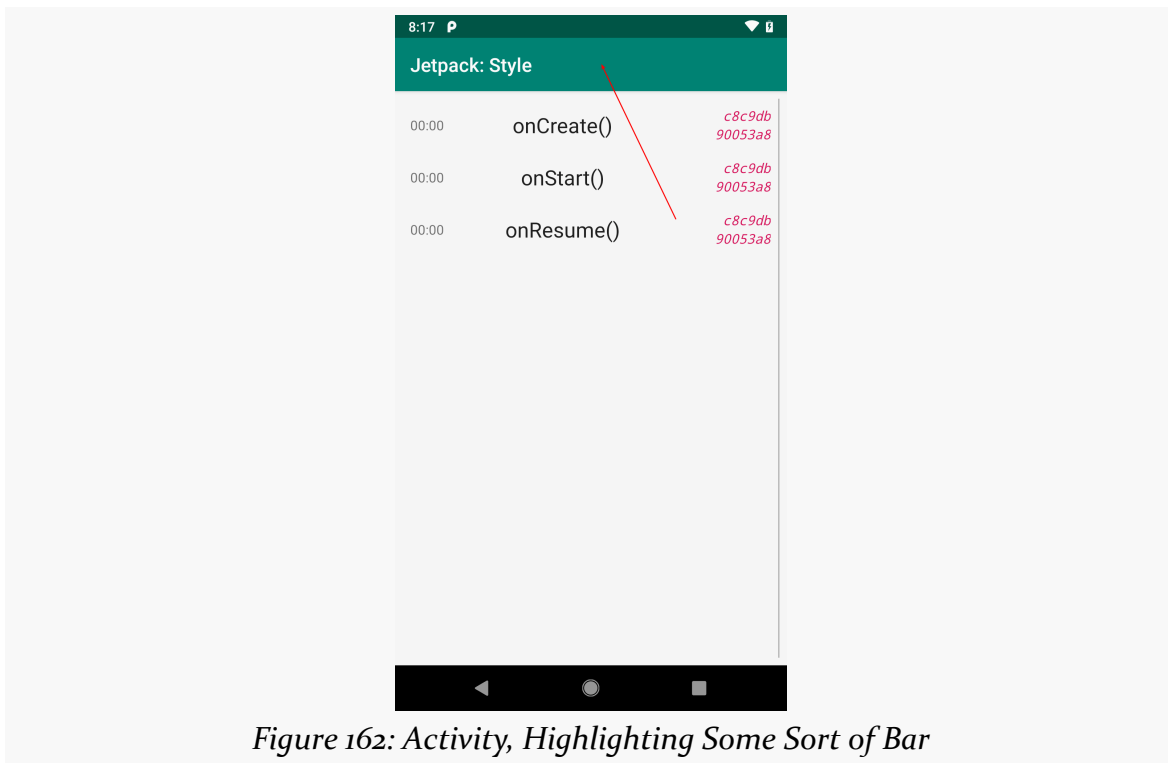


Figure 162: Activity, Highlighting Some Sort of Bar

To date, all it has done is show our title (app_name string resource). However, there is more that we can do with this bar, and we will explore some of that in this chapter.

Our first problem, though, in deciding what this thing is called.

So. Many. Bars.

Many aspects of Android have changed over the years. For example, few Android 1.x/2.x apps had this sort of bar at the top.

Sometimes, when Android changes, Google simply adds some new capability or design feature, and that's it. Sometimes, Google does one thing, then changes course and replaces it with something else. And, on occasion, Google makes so many changes that the result is quite a mess.

This bar is quite a mess.

Part of it is simply the name. Depending on who you ask, this bar could be referred to as:

- the action bar
- a toolbar
- an app bar
- a string of profanities, though usually these describe the developer's relationship with this bar and do not actually name the bar itself

Action Bar

In the beginning, we referred to this as the “action bar”. Activity and other classes had support for showing an action bar and doing things with it, such as dynamically changing the title shown in the bar.

Nowadays, “action bar” is more of a role than an actual thing. We can opt into using the action bar APIs, or we can achieve similar functionality without them.

Toolbar

The “action bar” terminology dominated Android 3.x and 4.x.

Android 5.0 debuted a Toolbar class. Initially, it was thought that the Toolbar was simply a bit of refactoring, giving us a widget that looked and worked like the action bar. The nice thing about Toolbar was that you could put one anywhere you wanted in your UI, making it simpler to add dedicated toolbars, such as for a rich-text editor.

Over time, Toolbar became the stock implementation of this sort of top-of-the-activity bar.

From a terminology standpoint, we have used “toolbar” as a descriptive term for quite some time, as “toolbar” is a widely-used term in desktop apps. A desktop app toolbar served a very similar role to the action bar in Android, and then the Toolbar in Android. So, you might see references to “toolbar buttons” in Android, which really refer to that sort of UI pattern, regardless of whether those buttons appear on an actual Toolbar or not.

App Bar

In 2015, Google debuted [“Material Design”](#). This is Google’s “design language” for mobile, Web, and desktop apps. It provides Google’s recommendations for what things should look and work like.

In Material Design, this bar is called the [app bar](#), because apparently Google likes coming up with new names for this.

The Material Design team has also created [“Material Components for Android”](#), which is a library that implements many of the UI patterns seen in Material Design that go beyond what modern versions of Android support directly. There, they have classes like `AppBarLayout` and `CollapsingToolbarLayout`, for trying to implement some of the specific Material Design recommendations regarding app bars, such as how they behave with respect to scrolling content in an activity.

Bars and This Book

From this point forward, the book will tend to use:

- Toolbar, in monospace, when referring to the actual Toolbar class
- “App bar”, when referring to the concept of this bar
- “Action bar”, when referring to that specific role
- “toolbar buttons”, when referring to the icons that can appear in this bar that the user can tap on to perform actions

(the book will use few profanities, no matter how appropriate they may be for aspects of Android app development)

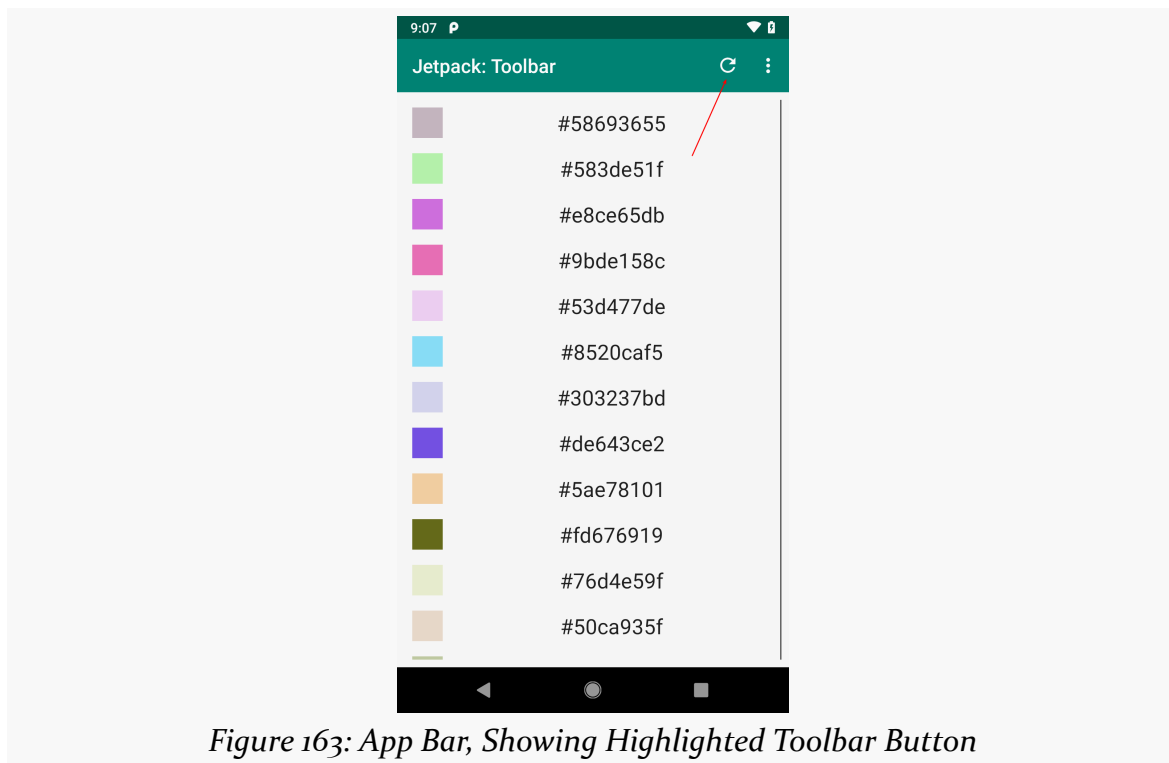
Bars Beyond These Bars

There are other bars in the Android UI that are unrelated to the app bar:

- The status bar is the thin strip across the top containing the time and various icons for device status
- The navigation bar is the strip across the bottom on some devices that provides the HOME button and — depending on Android version and situation — buttons for BACK and RECENTS (the latter of which brings up the overview screen)

Vector Drawables

Frequently, the app bar contains toolbar buttons, such as the refresh one shown below:



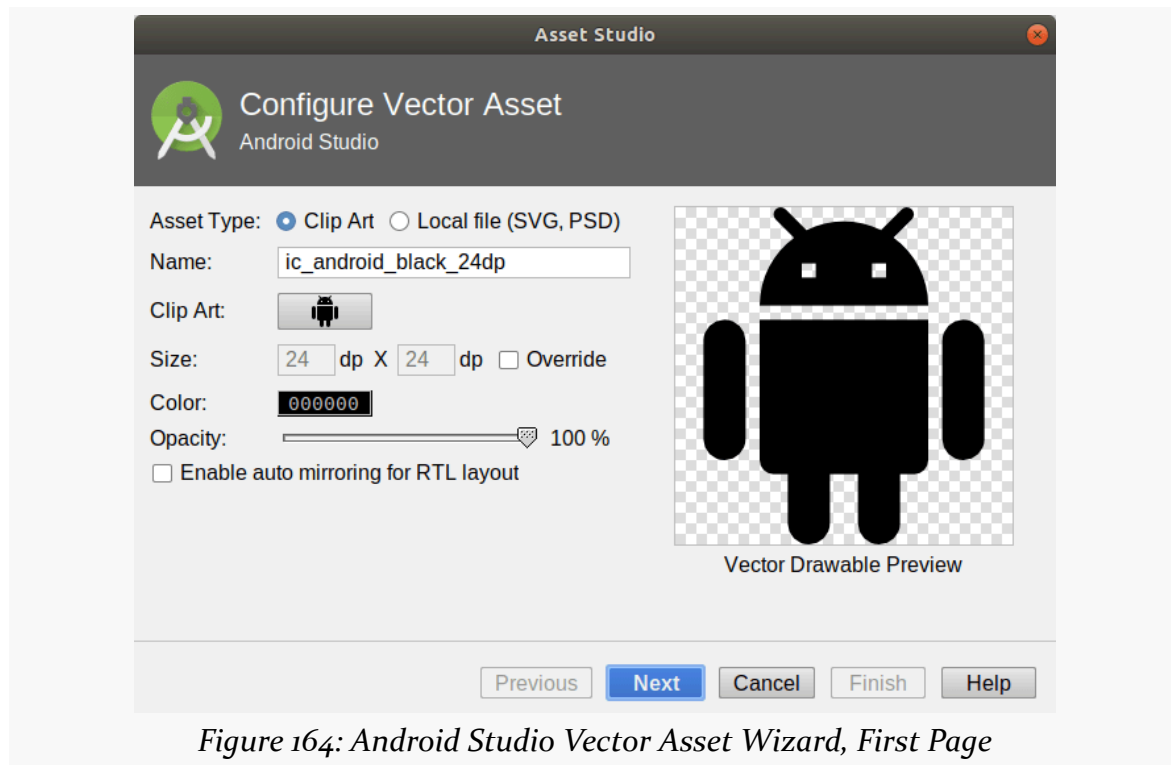
We could use PNG images for these. However, typically, icons like this one are implemented using vector drawables. These are drawn dynamically, so we do not need different versions of vector drawables for different screen densities, the way

that we do with PNGs. Plus, Google supplies us with a library of existing vector artwork that we can use, in addition to importing SVG files from elsewhere, such as from a graphic designer.

Starting the Vector Asset Wizard

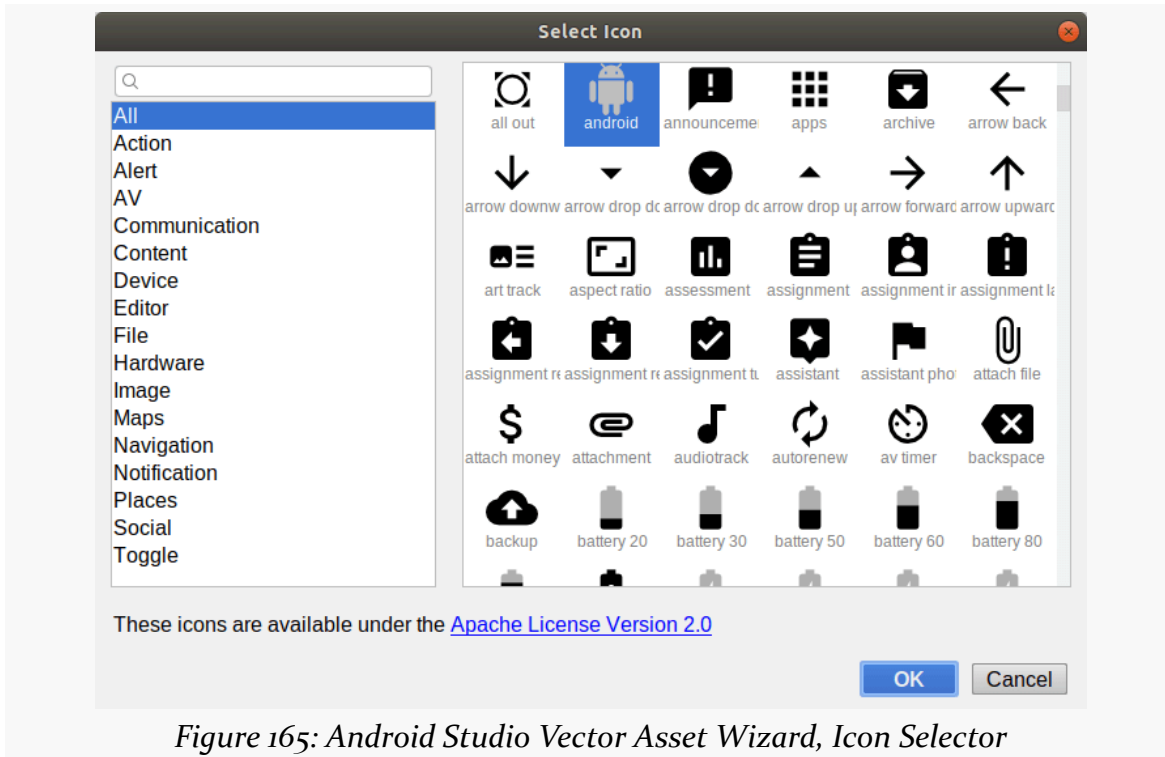
To add a vector drawable to your app — either from Google-supplied artwork or SVGs — you can use the Vector Asset Wizard. You can start this by right-clicking over anything in your module’s portion of the project tree in Android Studio, then choosing “New” > “Vector Asset” from the context menu.

This will bring up the Vector Asset Wizard on its first page:



Using Built-In Vector Artwork

By default, the wizard starts off with the “Asset Type” radio buttons set to “Clip Art”, allowing you to choose an icon from a Google-supplied library. The default icon is the Android mascot (“Bugdroid”), but you can click the button next to “Clip Art” to bring up the catalog of available artwork:



Here you can:

- Search via the search field in the upper left;
- Browse all of the icons, by choosing “All” in the list on the left and scrolling through the icons; or
- Browse a category of the icons, by choosing anything other than “All” in the list on the left and scrolling through the chosen category of icons

CONFIGURING THE APP BAR

Once you have identified the icon that you want, you can click OK to return to the wizard, with your icon selected:

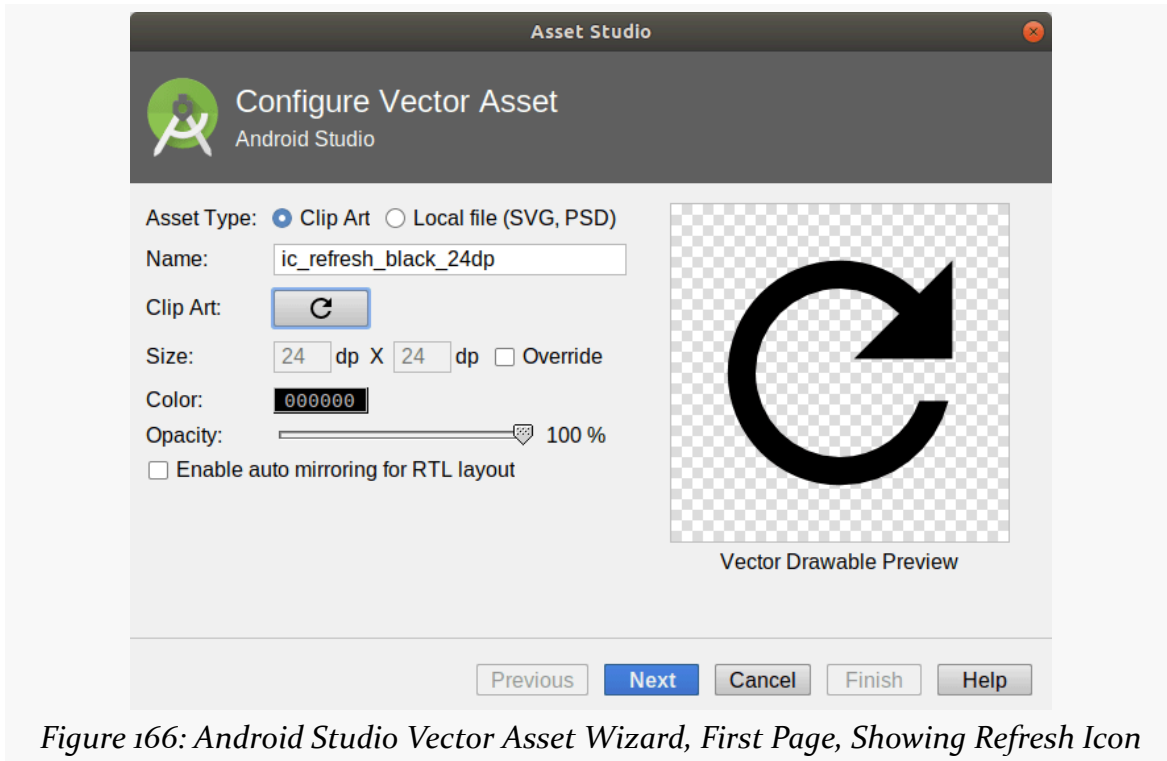


Figure 166: Android Studio Vector Asset Wizard, First Page, Showing Refresh Icon

An icon name will be filled in for you in the “Name” field, though you can change that if you wish. Similarly, you can override:

- The size of the icon (default is 24dp by 24dp)
- The color of the icon (default is black)
- The opacity of the icon, to control the color’s alpha channel (default is opaque)
- Whether the icon should be flipped on RTL screens or should remain unchanged

You can then click Next, followed by “Finish”, to add the vector drawable to the res/drawable/ directory of your module.

Importing SVGs

If you have a vector image in SVG or PSD format from a graphic designer, you can try to use it. Android’s vector drawables implement a subset of SVG, and some SVGs will

CONFIGURING THE APP BAR

be too complex for Android to support.

To use an existing SVG file, rather than browse the existing icons, choose “Local file (SVG, PSD)” in the “Asset Type” radio buttons in the first page of the Vector Asset Wizard:

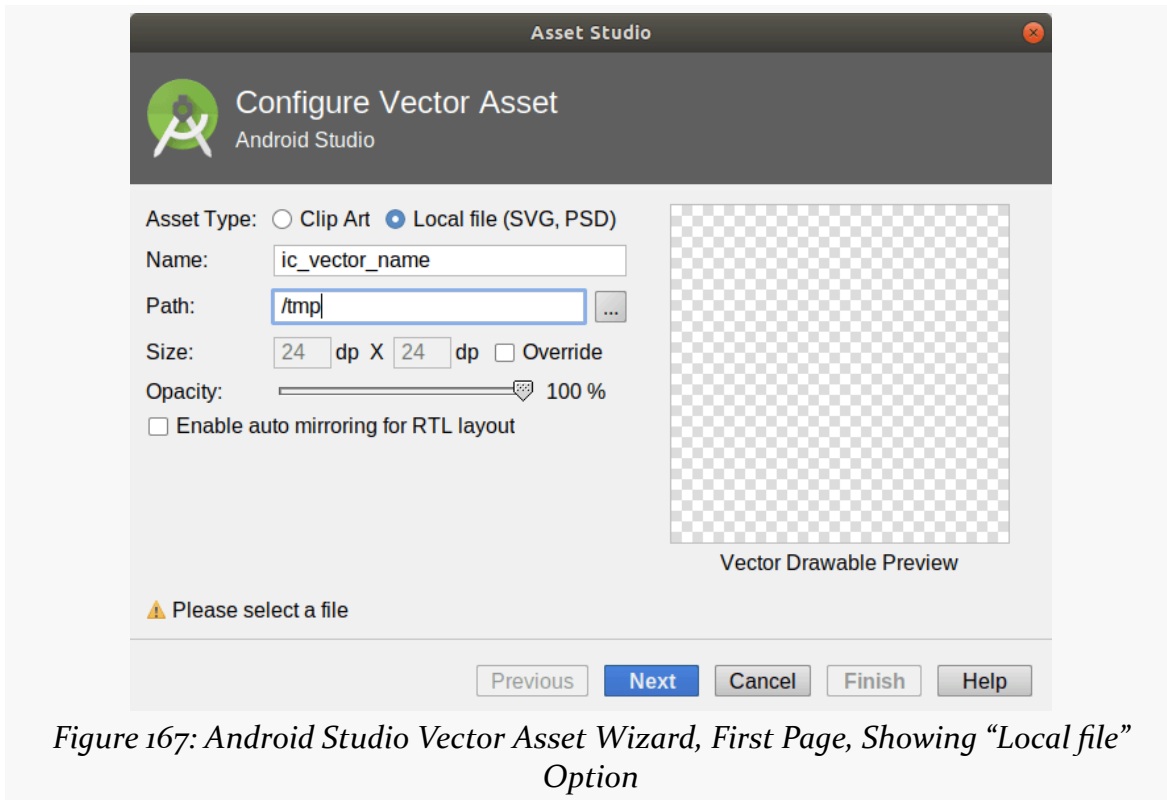


Figure 167: Android Studio Vector Asset Wizard, First Page, Showing “Local file” Option

CONFIGURING THE APP BAR

You can click the “...” button next to the “Path” field to browse for your SVG or PSD file. After choosing the file, the Vector Asset Wizard will load it and give you some idea of whether or not it will work. Specifically, if the preview does not look promising, or you get warnings, that particular file may not work well as a vector drawable:

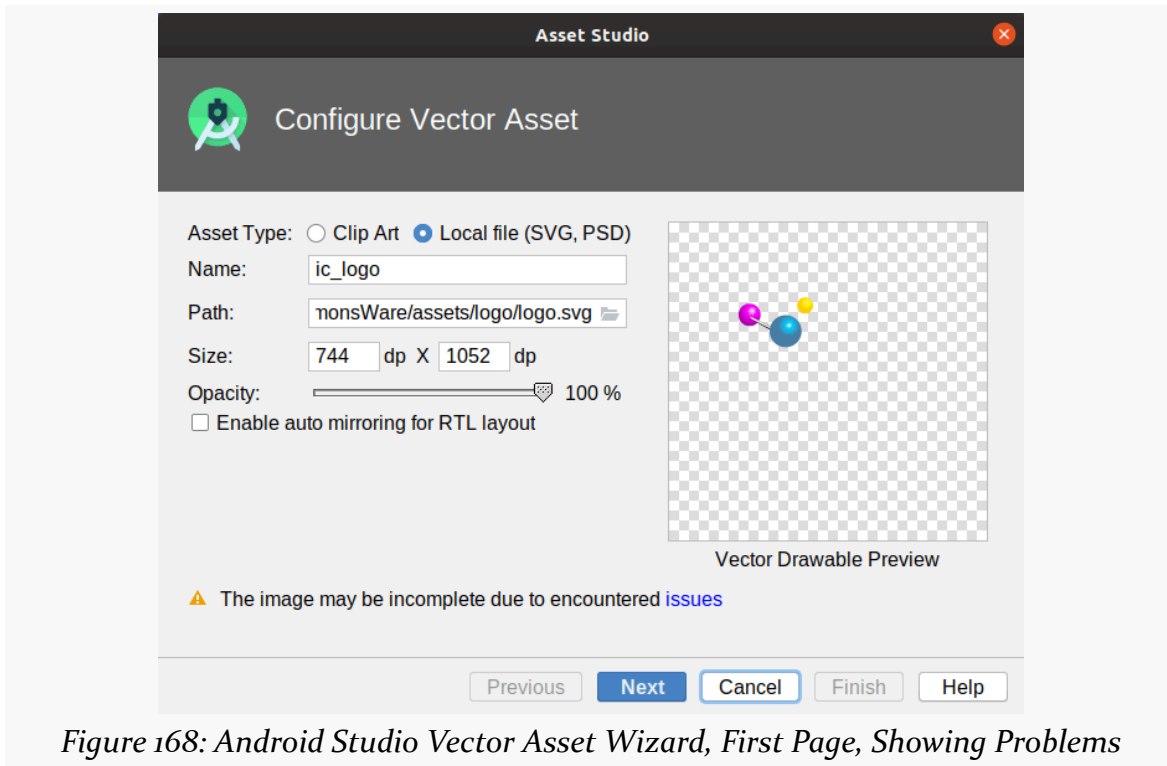


Figure 168: Android Studio Vector Asset Wizard, First Page, Showing Problems

Beyond that, though, you can override the size, opacity, and RTL settings, then proceed through the rest of the wizard to create the vector drawable.

Menu Resources

The next step in configuring the app bar is to set up a menu resource to reflect the clickable items that should appear in the app bar, such as toolbar buttons.

Why “Menu”?

Back in the dawn of Android time, referred to by some as “2007”, we had options menus. These would rise up from the bottom of the screen based on the user pressing a MENU key:



Figure 169: Legacy Options Menu

When Google introduced the action bar pattern with Android 3.0 in 2011, they used the old options menu system and simply changed its UI. So, you will see references to an “options menu” in our work with the action bar, as this is all based on that original 2011 implementation of the action bar.

Similarly, the options menu was populated using a menu resource. The action bar adopted the menu resource and extended it for its own purposes. Today, Toolbar continues to use menu resources, even though we might not think of our work involving a “menu” anymore.

Defining Menu Resources

Menu resources are “first-class” resources, like layouts, drawables, and mipmaps. They get their own dedicated `res/menu/` directory and have their own dedicated

Android Studio editor.

Creating the `res/menu/` Directory

However, when you create a brand-new Android Studio project, you may not have a `res/menu/` directory in your module.

Since this is an ordinary directory in an ordinary filesystem, you can create this directory by any means that you like, such as your development machine’s “file manager”, or via a `mkdir` command on the command line.

Inside of Android Studio, you have two main options for creating this directory. The simple one is to right-click over the `res/` directory of your module, then choose “New” > “Directory” from the context menu. This will pop up a dialog where you can provide your directory name — if you fill in `menu` and accept the dialog, it will create the `menu/` subdirectory under `res/` for you.

The more elaborate solution is to right-click over any directory in your module and choose “New” > “Android Resource Directory” from the context menu. This brings up a “New Resource Directory” dialog:

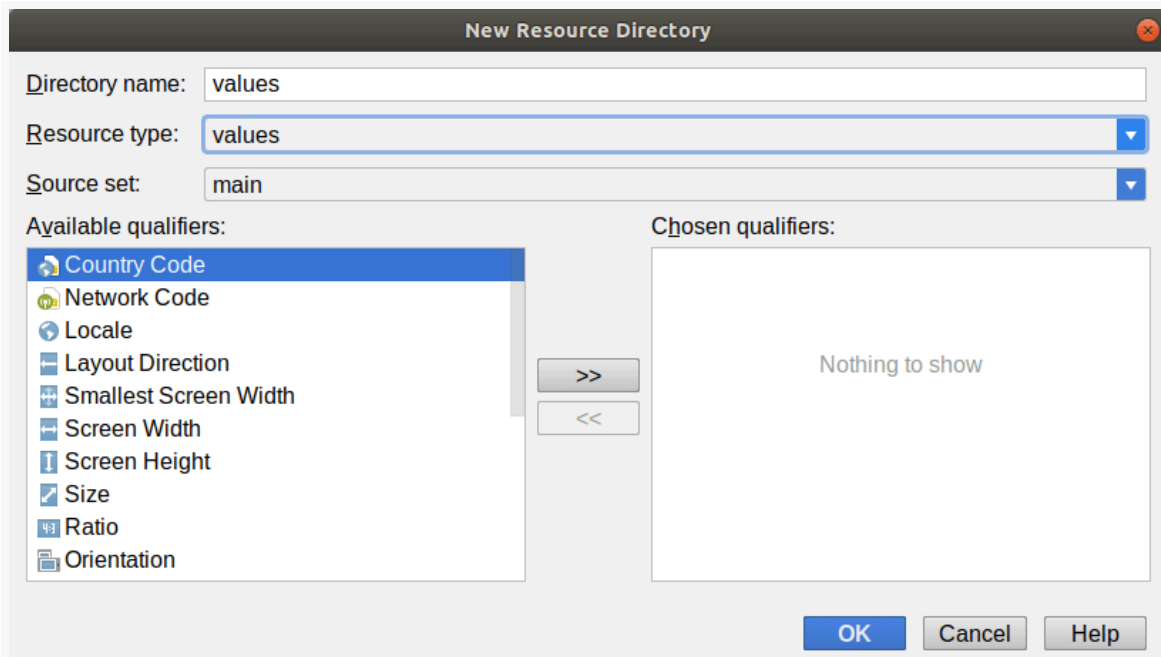


Figure 170: Android Studio New Resource Directory Dialog

There, you can choose “menu” in the “Resource type” drop-down, then click “OK” to create the `res/menu/` directory. This dialog also allows you to create directories for resource sets via the “Available qualifiers” list, and it has a few other “bells and whistles”, but in the end, it just creates a directory.

Creating the Menu Resource

Once you have the `res/menu/` directory, you can create an empty menu resource.

In Android Studio, that is a matter of:

1. Right-clicking over your `res/menu/` directory
2. Choosing “New” > “Menu resource file” from the context menu, to bring up a simple “New Menu Resource File” dialog
3. Entering the base name of the resource (e.g., `actions`) into the dialog
4. Clicking “OK” to close the dialog and create the nearly-empty resource

You will wind up with a menu resource based on your chosen name (e.g., `actions.xml`) with an empty `<menu>` root element:

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

Adding Menu Items

The menu resource editor works like the graphical layout editor. You can either work with the XML directly or use the drag-and-drop GUI builder. The drag-and-drop GUI builder has the same basic structure as does the layout editor, with a palette, a component tree, a preview area, and an attributes pane:

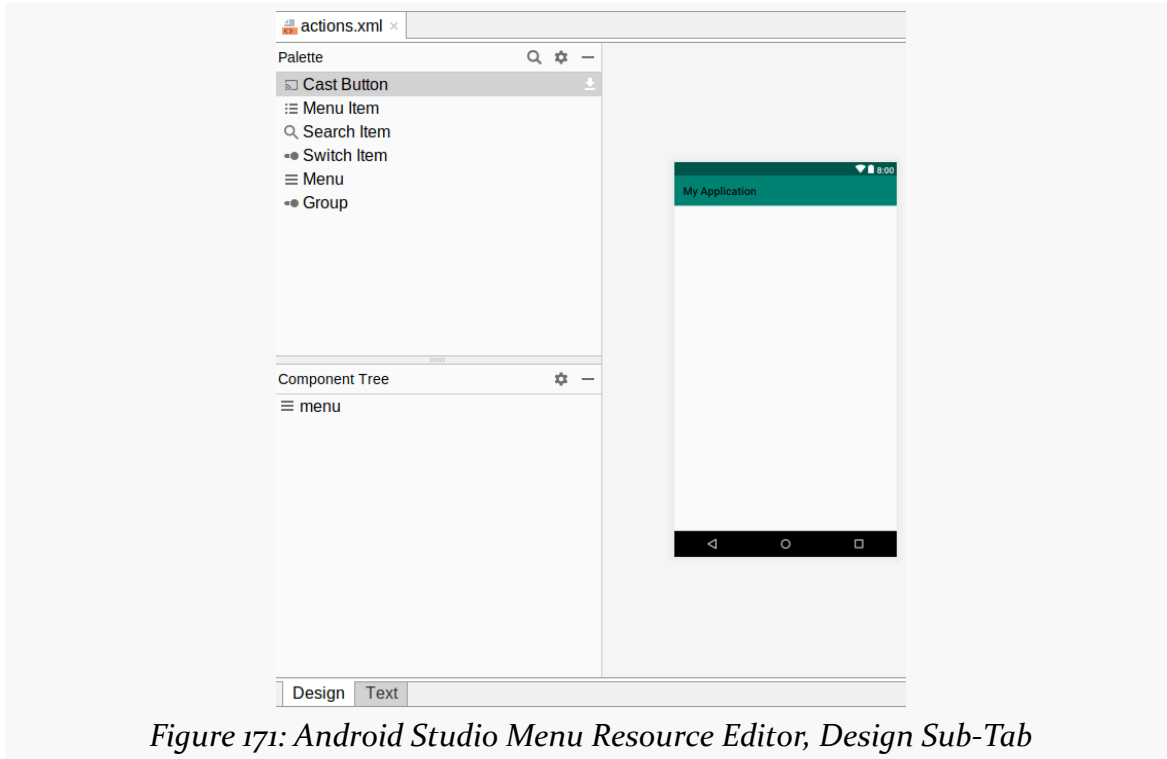


Figure 171: Android Studio Menu Resource Editor, Design Sub-Tab

CONFIGURING THE APP BAR

While there are several items in the “Palette” tool, mostly you will be working with “Menu Item” elements. You can drag them from the “Palette” into the “Component Tree” or preview area to add them to the menu resource:

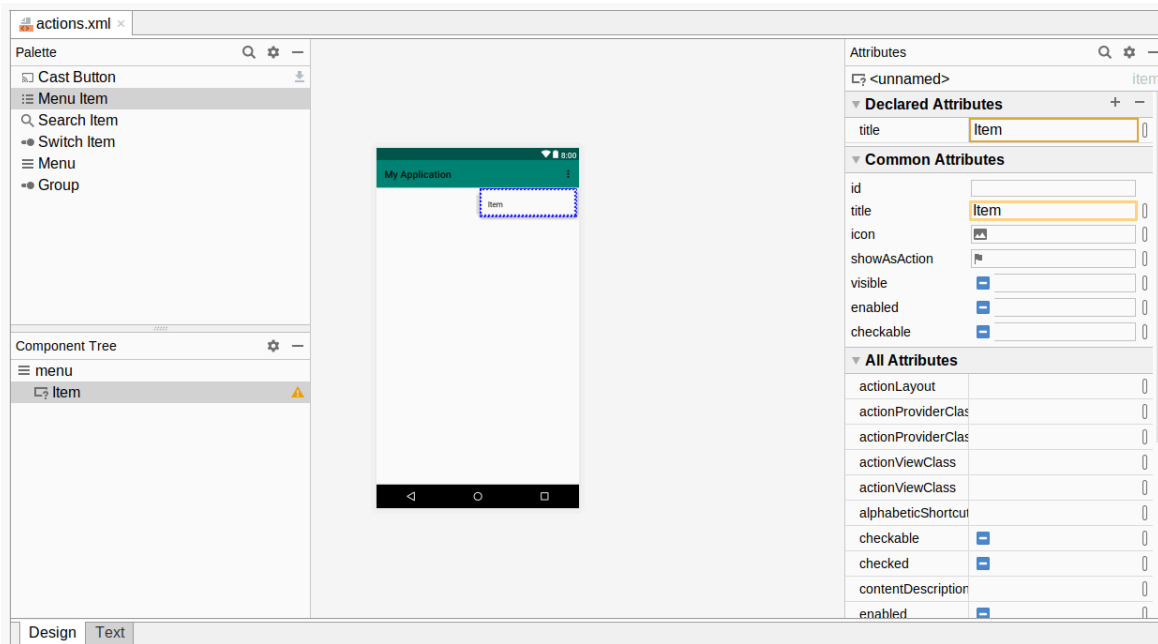


Figure 172: Android Studio Menu Resource Editor, With Added Menu Item

The “Attributes” pane will then allow you to manipulate the menu item’s attributes — we will explore this more [in an upcoming section](#).

Using Toolbar Directly

So, let’s see how we can use vector drawables and menu resources to add interactive elements to the app bar. First, we will look at how to do this using `Toolbar` directly as a widget.

The `Toolbar` sample module in the [Sampler](#) and [SamplerJ](#) projects are a variation on the show-a-list-of-random-colors `ViewModel` sample from earlier in the book. Most of the changes are tied to using `Toolbar`, but if you examine the full modules, you will see that this sample also blends in the instance state management — using `ViewModelProvider.Factory` — that we saw in some of show-a-list-of-lifecycle-events samples.

Adding the Widget

We can add a `Toolbar` to our layout, positioning it wherever we want. Typically, it appears at the top of the activity, but that is not a requirement.

So, the `Toolbar` project edition of the `activity_main` layout has a `Toolbar` above the `RecyclerView`, wrapped in the `ConstraintLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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=".MainActivity">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:background="?attr/colorPrimary"
        app:theme="?attr/actionBarPopupTheme"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/items"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:padding="@dimen/content_padding"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@id/toolbar" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [Toolbar/src/main/res/layout/activity_main.xml](#))

Usually, the app bar is set to be flush with the status bar and the sides of the screen. So, whereas the `ViewModel` sample had 8dp of padding in the `ConstraintLayout`, this sample moves that padding to the `RecyclerView`, so the `Toolbar` is not inset from the edges.

CONFIGURING THE APP BAR

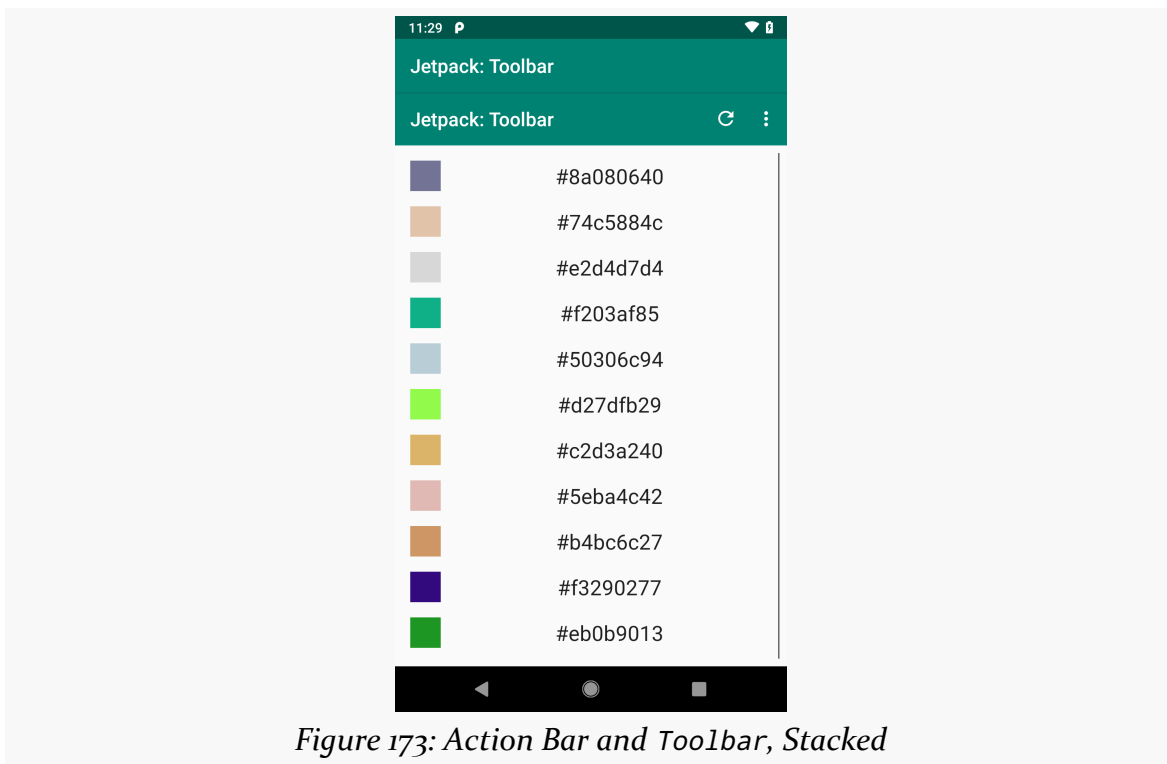
The Toolbar has attributes that mostly set up the size and position. A simple Toolbar can use `wrap_content` for the height, and it will be sized appropriately for toolbar buttons and such. `android:background` allows you to specify the background color, and here we defer to the theme and use whatever we have set there as `colorPrimary` (`?attr/colorPrimary`). We will explore the `app:theme` attribute more in the next section.

Tailoring the Theme

The theme used by prior sample projects in this book was based on `Theme.AppCompat.Light.DarkActionBar`. As the name suggests, this adds an action bar to the top of the activity, with an eye towards it having a dark color, so text will be shown in white for a good contrast.

The problem is that we do not want an action bar. We have our own Toolbar that we want to show instead.

The theme has no way of knowing this, and so by default, we would wind up with two app bars, stacked on top of each other:



CONFIGURING THE APP BAR

This is... not good.

There is another base theme, `Theme.AppCompat.Light.NoActionBar`, which skips the theme-supplied action bar, leaving it up to us. So, the Toolbar sample modules use that base theme instead:

```
<resources>

  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
    <item name="actionBarPopupTheme">@style/PopupOverlay</item>
  </style>

  <style name="PopupOverlay" parent="ThemeOverlay.AppCompat.Dark.ActionBar">
    <item name="iconTint">@android:color/white</item>
  </style>

</resources>
```

(from [Toolbar/src/main/res/values/styles.xml](#))

The next problem is that the stock look of a Toolbar assumes that the background color will be light, and so we want dark text and icons on it. In our case, the background color is relatively dark, and so we would prefer light text and icons. This is a bit tricky to set up. One recipe is:

- Define a style resource that inherits from `ThemeOverlay.AppCompat.Dark.ActionBar` (here called `PopupOverlay`)
- In that style resource, define the `iconTint` to be whatever color you want — in this case, we are using `@android:color/white` to pull in a framework-defined white color
- In the theme for the activity (`AppTheme` in our case), define an `actionBarPopupTheme` attribute and have it point to the style resource that you just created
- Give the Toolbar an `app:theme` attribute that delegates to the theme's `actionBarPopupTheme` (`app:theme="?attr/actionBarPopupTheme"`)

`ThemeOverlay.AppCompat.Dark.ActionBar` will give us light text, and white for the `iconTint` will give us light icons.

Defining the Menu Resource

The sample app has a menu resource, `res/menu/actions.xml`, that contain interactive elements that will go into the Toolbar:

```
<?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/refresh"
    android:title="@string/menu_refresh"
    android:icon="@drawable/ic_refresh_black_24dp"
    app:showAsAction="ifRoom" />
  <item
    android:id="@+id/about"
    android:title="@string/menu_about"
    app:showAsAction="never" />
</menu>
```

(from [Toolbar/src/main/res/menu/actions.xml](#))

CONFIGURING THE APP BAR

In the graphical menu editor, we see two children — refresh and about — of the root menu in the “Component Tree”:

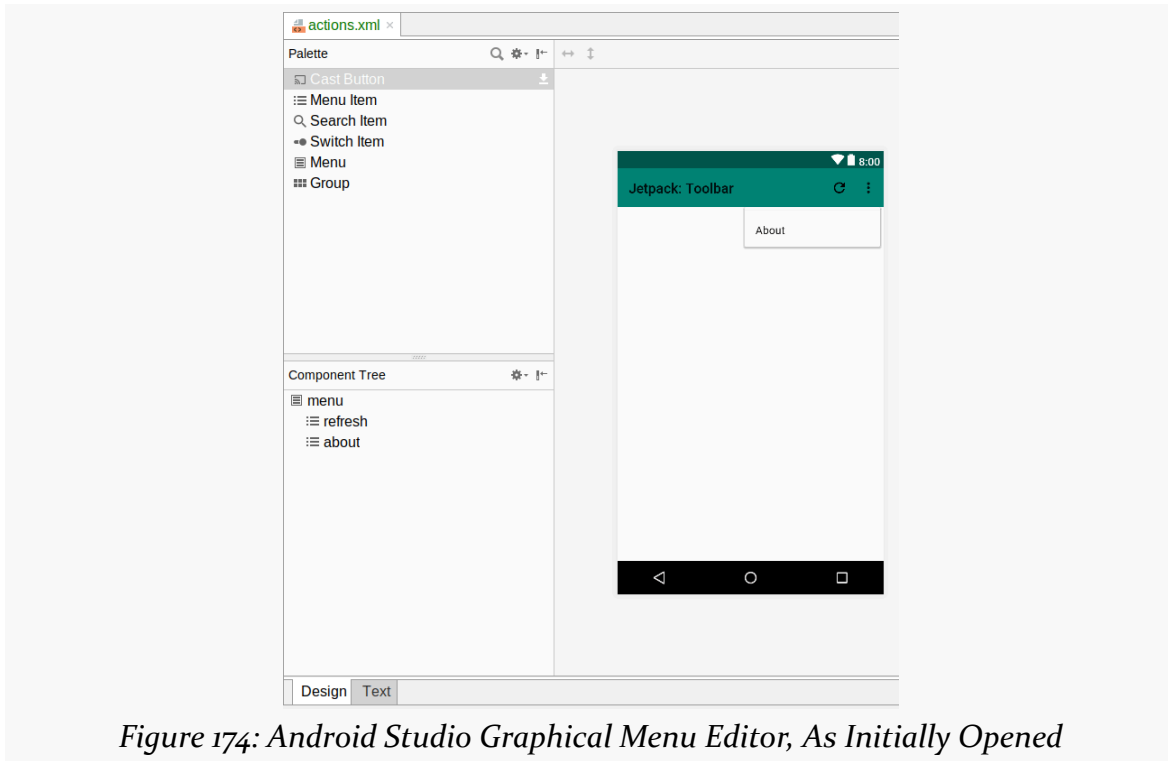


Figure 174: Android Studio Graphical Menu Editor, As Initially Opened

The preview area shows these items as well, though they appear differently due to the way that each of those items is configured, as we will see.

Hey, Why Is the Preview Showing Dark Text and Icons?

In the above screenshot, the title (“Jetpack: Toolbar”) and the icons in the Toolbar show up dark. The graphical menu editor attempts to use our app’s theme, but it has limits, and in this case it is not picking up some of the changes that [we made to the theme](#).

In general, consider the Android Studio preview options to be approximations of what you will see when you run the app on an emulator or device.

Refresh

One of our two menu items is called refresh, and we will use it to allow the user to

CONFIGURING THE APP BAR

come up with a new random list of colors:

```
<item
  android:id="@+id/refresh"
  android:title="@string/menu_refresh"
  android:icon="@drawable/ic_refresh_black_24dp"
  app:showAsAction="ifRoom" />
```

(from [Toolbar/src/main/res/menu/actions.xml](#))

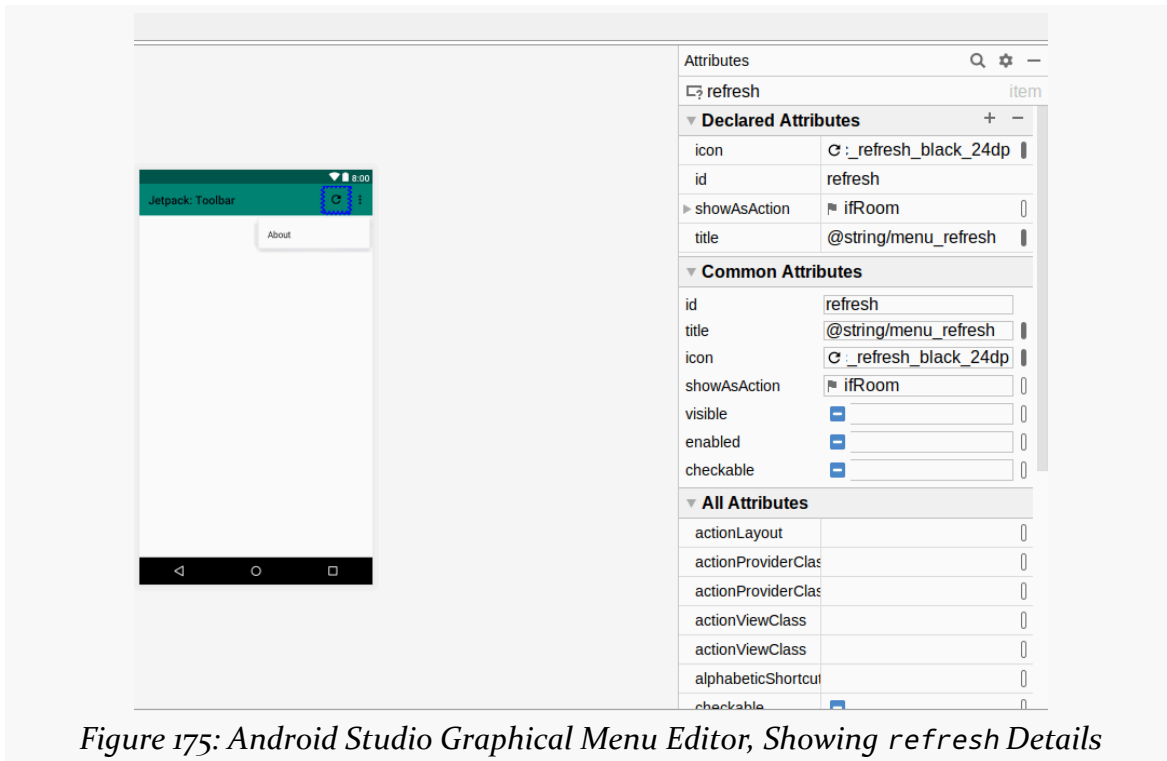


Figure 175: Android Studio Graphical Menu Editor, Showing refresh Details

Each interactive element in our Toolbar will have an `<item>` element inside of the root `<menu>` element of our menu resource. Items always have:

- An ID, using the same `android:id` system that we use for widget IDs
- A title, via the `android:title` attribute, usually pointing to a string resource

Items *usually* have a `showAsAction` attribute. For AppCompatActivity-based activities, that should be `app:showAsAction` — if you see code with `android:showAsAction`, that is a menu resource for use with the native framework action bar or Toolbar, not the AppCompatActivity-compatible Toolbar that we use from AndroidX. `showAsAction` has three major options:

CONFIGURING THE APP BAR

- always, to say that we *really* want this item to be shown as a toolbar button
- `ifRoom`, to say that we would prefer it be shown as a toolbar button, but that is not essential
- `never`, to say that this item is unimportant and does not need a dedicated button in the Toolbar

Items that go into the Toolbar that do not wind up with toolbar buttons go into “the overflow menu”. If you have used Android apps that have a vertical ellipsis (“...”) icon in their app bars, where a menu pops open when you tap it... that’s the overflow menu. In our case, the refresh item has `ifRoom`, so if there is space for it to have a toolbar button, it will have one, otherwise it will go into the overflow.

For items that might be toolbar buttons, usually you will want to provide an `android:icon` attribute with a pointer to a drawable resource. Here, we are using a refresh icon from the Vector Asset Wizard, using a stock refresh vector asset that is supplied by Google.

About

Our about menu item is not nearly as important, so we set it to have `never` for `showAsAction` and skip the `android:icon` attribute:

```
<item
    android:id="@+id/about"
    android:title="@string/menu_about"
    app:showAsAction="never" />
```

CONFIGURING THE APP BAR

(from [Toolbar/src/main/res/menu/actions.xml](#))

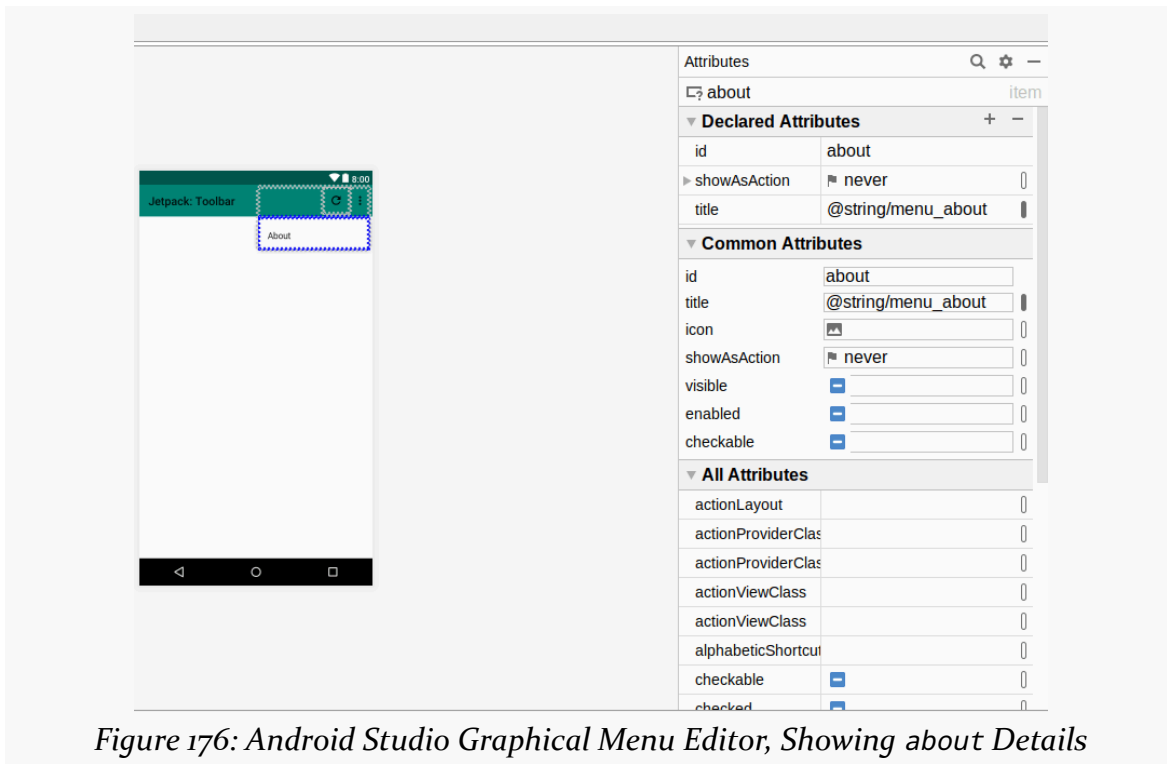


Figure 176: Android Studio Graphical Menu Editor, Showing about Details

Populating the Toolbar

We have a Toolbar in our layout. We have a menu resource. We now need to tie the two together. That requires a bit of code in `onCreate()` of our `MainActivity`, whether that is in Java:

```
binding.toolbar.setTitle(R.string.app_name);
binding.toolbar.inflateMenu(R.menu.actions);
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/MainActivity.java](#))

...or Kotlin:

```
binding.toolbar.apply {
    setTitle(R.string.app_name)
    inflateMenu(R.menu.actions)
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/sampler/toolbar/MainActivity.kt](#))

(the Kotlin will appear to be missing a closing brace, but that is because there is

CONFIGURING THE APP BAR

more code in this `apply()` that we will see in the next section)

First, we get access to the `Toolbar`, either via `findViewById()` in Java or by using synthetic accessors in Kotlin. Then, we call two functions on the `Toolbar`:

- `setTitle()` sets the title text — when we use a `Toolbar` manually like this, we need to provide that value ourselves
- `inflateMenu()`, where we pass it an ID of our menu resource (`R.menu.actions`)

Under the covers, `inflateMenu()` uses a `MenuInflater` to convert our menu resource XML into `Menu` and `MenuItem` objects, which `Toolbar` will use to configure the `Toolbar` contents and show our toolbar button and overflow menu:

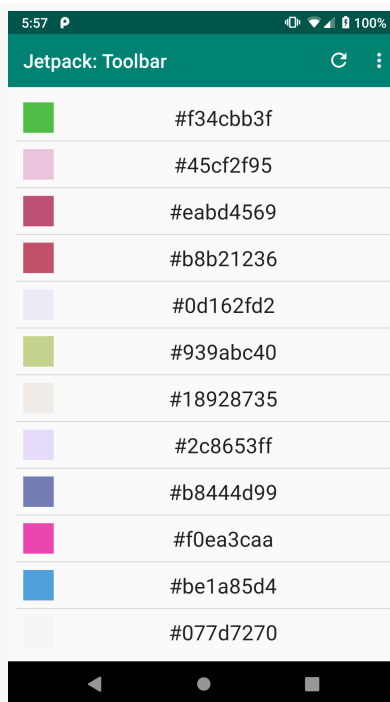


Figure 177: Toolbar Sample, As Initially Launched on Android 9.0

CONFIGURING THE APP BAR

The overflow menu appears when you tap the “...” button in the Toolbar:

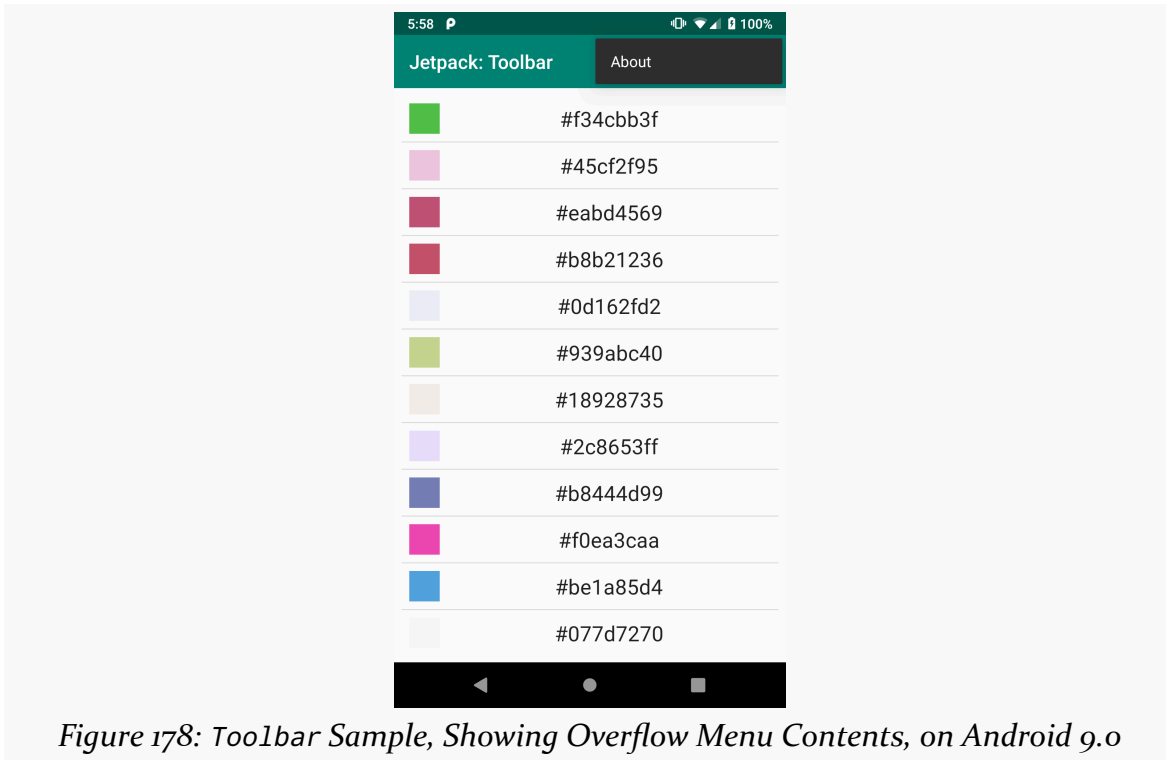


Figure 178: Toolbar Sample, Showing Overflow Menu Contents, on Android 9.0

Responding to Events

While our toolbar button and overflow menu are pretty, they are useless unless we arrange to find out when the user clicks on our refresh and about items.

For that, we call `setOnMenuItemClickListener()` on the Toolbar:

```
binding.toolbar.setOnMenuItemClickListener(item -> {
    if (item.getItemId() == R.id.refresh) {
        vm.refresh();
        adapter.submitList(vm.numbers);
        return true;
    }
    else if (item.getItemId() == R.id.about) {
        Toast.makeText(MainActivity.this, R.string.msg_toast,
            Toast.LENGTH_LONG).show();
        return true;
    }
    else {
```

CONFIGURING THE APP BAR

```
        return false;
    }
    });
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/MainActivity.java](#))

```
binding.toolbar.apply {
    setTitle(R.string.app_name)
    inflateMenu(R.menu.actions)

    setOnMenuItemClickListener { item ->
        when (item.itemId) {
            R.id.refresh -> {
                vm.refresh()
                colorAdapter.submitList(vm.numbers)
                true
            }
            R.id.about -> {
                Toast.makeText(
                    this@MainActivity,
                    R.string.msg_toast,
                    Toast.LENGTH_LONG
                ).show()
                true
            }
            else -> false
        }
    }
}
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/MainActivity.kt](#))

Our lambda expression is passed the MenuItem object corresponding to the <item> from our menu resource that the user tapped on. We can call getItemId() to determine which <item> it was.

If they clicked refresh, we call a refresh() function on the revised ColorViewModel that generates a fresh set of colors:

```
package com.commonsware.jetpack.samplerj.toolbar;

import java.util.ArrayList;
import java.util.Random;
import androidx.lifecycle.SavedStateHandle;
import androidx.lifecycle.ViewModel;

public class ColorViewModel extends ViewModel {
```

CONFIGURING THE APP BAR

```
private static final String STATE_NUMBERS = "numbers";
ArrayList<Integer> numbers;
private final SavedStateHandle state;

public ColorViewModel(SavedStateHandle state) {
    this.state = state;
    numbers = state.get(STATE_NUMBERS);

    if (numbers == null) {
        numbers = buildItems();
    }
}

void refresh() {
    numbers = buildItems();
    state.set(STATE_NUMBERS, numbers);
}

private ArrayList<Integer> buildItems() {
    Random random = new Random();

    ArrayList<Integer> result = new ArrayList<>(25);

    for (int i = 0; i < 25; i++) {
        result.add(random.nextInt());
    }

    return result;
}
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/ColorViewModel.java](#))

```
package com.commonsware.jetpack.sampler.toolbar

import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import java.util.*

private const val STATE_NUMBERS = "numbers"

class ColorViewModel(private val state: SavedStateHandle) : ViewModel() {
    private val random = Random()
    var numbers = state.get<List<Int>>(STATE_NUMBERS) ?: buildItems()

    fun refresh() {
        numbers = buildItems()
        state.set(STATE_NUMBERS, numbers)
    }
}
```

CONFIGURING THE APP BAR

```
}  
  
private fun buildItems() = List(25) { random.nextInt() }  
}
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/sampler/toolbar/ColorViewModel.kt](#))

We then update the ColorAdapter with the new colors, and it updates the list.

If the user clicked about, we show a Toast — in a real app, an “About” item would bring up some screen that contains a copyright notice, license terms, version information, and so on.

The lambda expression needs to return true if the item click was handled by the lambda, or false if the item was not recognized for one reason or another.

Using Toolbar as the Action Bar

There is also an option for us to take the Toolbar from our layout and tell AppCompatActivity to use it as the action bar. This has value in some cases, such as when we start working with fragments in [an upcoming chapter](#).

The ActionBar modules in the [Sampler](#) and [SamplerJ](#) projects are almost the same as the Toolbar modules. We use the same layout, the same menu resource, and the same theme. The difference lies in the Java/Kotlin code for setting up the Toolbar and responding to events.

Registering the Toolbar

To indicate that a Toolbar should serve in the role of the action bar, call `setSupportActionBar()` on your AppCompatActivity, supplying the Toolbar:

```
setSupportActionBar(binding.toolbar);
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/samplerj/actionbar/MainActivity.java](#))

```
setSupportActionBar(binding.toolbar)
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/sampler/actionbar/MainActivity.kt](#))

The action bar automatically gets the title from its activity’s `android:label` manifest attribute, so you may not need to call `setTitle()` yourself on the Toolbar.

Populating the Action Bar

Rather than call `inflateMenu()` on the Toolbar, the action bar reuses the old `onCreateOptionsMenu()` callback function from the days of options menus. Your `AppCompatActivity` subclass — such as `MainActivity` in the ActionBar samples — will need to override this:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    return super.onCreateOptionsMenu(menu);
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/samplerj/actionbar/MainActivity.java](#))

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.actions, menu)

    return super.onCreateOptionsMenu(menu)
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/sampler/actionbar/MainActivity.kt](#))

In `onCreateOptionsMenu()`, you:

- Get a `MenuInflater` by calling `getMenuInflater()` on the activity
- Call `inflate()` on the `MenuInflater`, passing in your menu resource ID (`R.menu.actions`) and the `Menu` object supplied to `onCreateOptionsMenu()`
- Chain to the superclass' implementation of `onCreateOptionsMenu()` and return its results

Responding to Events

Similarly, instead of calling `setOnMenuItemClickListener()` on the Toolbar to find out when the user clicks on items, you override the legacy `onOptionsItemSelected()` function:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.refresh) {
        vm.refresh();
        adapter.submitList(vm.numbers);
        return true;
    }
    return false;
}
```

CONFIGURING THE APP BAR

```
}  
else if (item.getItemId() == R.id.about) {  
    Toast.makeText(MainActivity.this, R.string.msg_toast,  
        Toast.LENGTH_LONG).show();  
    return true;  
}  
else {  
    return super.onOptionsItemSelected(item);  
}  
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/samplerj/actionbar/MainActivity.java](#))

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {  
    return when(item.itemId) {  
        R.id.refresh -> {  
            vm.refresh()  
            colorAdapter.submitList(vm.numbers)  
            true  
        }  
        R.id.about -> {  
            Toast.makeText(  
                this@MainActivity,  
                R.string.msg_toast,  
                Toast.LENGTH_LONG  
            ).show()  
            true  
        }  
        else -> super.onOptionsItemSelected(item)  
    }  
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/sampler/actionbar/MainActivity.kt](#))

This function does the same basic thing as the lambda expression supplied to `setOnMenuItemClickListener()`:

- Get the item ID from the `MenuItem`
- Compare it to expected values (e.g., a when in Kotlin)
- Handle the events for those expected values (e.g., showing a Toast)
- Return true if the event was handled

In the unrecognized-item scenario, since we chained to the superclass in `onCreateOptionsMenu()`, we chain to the superclass in `onOptionsItemSelected()`. That way, if our superclass is contributing items to the action bar, we will both add those items (`onCreateOptionsMenu()`) and handle their click events

(onOptionsItemSelected()).

Visually, this sample is indistinguishable from the Toolbar one, other than using a different `app_name` string resource. Functionally, it is merely a matter of where and how you apply the menu resource and deal with the results.

Having Fun at Bars

There are many other things that you can do with an app bar, just with the Android SDK, such as:

- Have checkable menu items, particularly for the overflow, that the user can check and uncheck
- Add other sorts of widgets to it, such as search fields and custom menus
- Have a “contextual action bar” that overlays the regular app bar with options tied to the UI state, such as providing actions for manipulating the current selected item(s) in a list
- Hide and show the app bar, so it does not take up screen space all the time (e.g., in a video player)

Other libraries can offer other features. For example, Google publishes the [Material Components for Android](#) library, which offers extensions like:

- Bottom app bars (i.e., an app bar that appears on the bottom, rather than the top)
- Collapsing toolbars (i.e., ones that appear super-sized at the outset, then collapse as the user scrolls through content)

Implementing Multiple Activities

All of the apps that we have seen so far in this book have had a single screen's worth of UI. Many apps are somewhat more complicated than that, where we need to have lots of screens, such as:

- Showing a list of stuff
- Showing the details of a particular item out of that list
- Editing the details of a particular item, or adding a new one
- Settings to configure how the app behaves
- And so on

In the world of Jetpack, there are two main approaches for adding multiple screens. In this chapter, we will look at one: having more than one activity. In [the next chapter](#), we will explore the other option: using fragments.

Multiple Activities, and Your App

There are at least two scenarios where your app may have to deal with multiple activities.

The first is the one outlined above: you want to have different screens for handling different bits of app functionality. In that case, you are the one writing the additional activities, and you will be the one to choose when those activities get displayed (e.g., bring up the details when the user taps on an item in the list). In this case, you have other options, such as the fragments mentioned above. But a lot of older Android projects will take the approach of having an activity for each distinct screen.

The second is when you have some piece of data that you want some other app to

process. For example, you might have a URL to a Web page that you obtained from somewhere, and you want the user to view that Web page in their browser. In this case, you are not writing the Web browser (probably), but you are still deciding when that Web browser activity gets displayed. So, we need a way to start up another app, and that comes in the form of starting an activity from that app.

A less-common scenario is the inverse of the previous one. Suppose you *are* writing a Web browser. You want other apps to be able to hand you URLs, so you can display those Web pages to the user. In this case, you are writing the activity, but you may not be the one deciding when that activity gets displayed.

This chapter will focus on the first two of those scenarios, as they are the most common, though we will briefly cover how you support the third scenario.

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.

Given a module 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.

Defining the Class and Resources

To create your second (or third or whatever) activity, you first need to create the Java or Kotlin class. You need to create a new source file, containing a public class that extends `Activity` (or `AppCompatActivity`, etc.). You have two basic ways of doing this:

- Just create the class yourself
- Use the Android Studio new-activity wizard

To use the Android Studio new-activity wizard, right-click a package (e.g., `com.commonware.jetpack.sampler.activities` in the project tree, and go into the “New” > “Activity” portion of the context menu. This will give you a submenu of available activity templates.

IMPLEMENTING MULTIPLE ACTIVITIES

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:

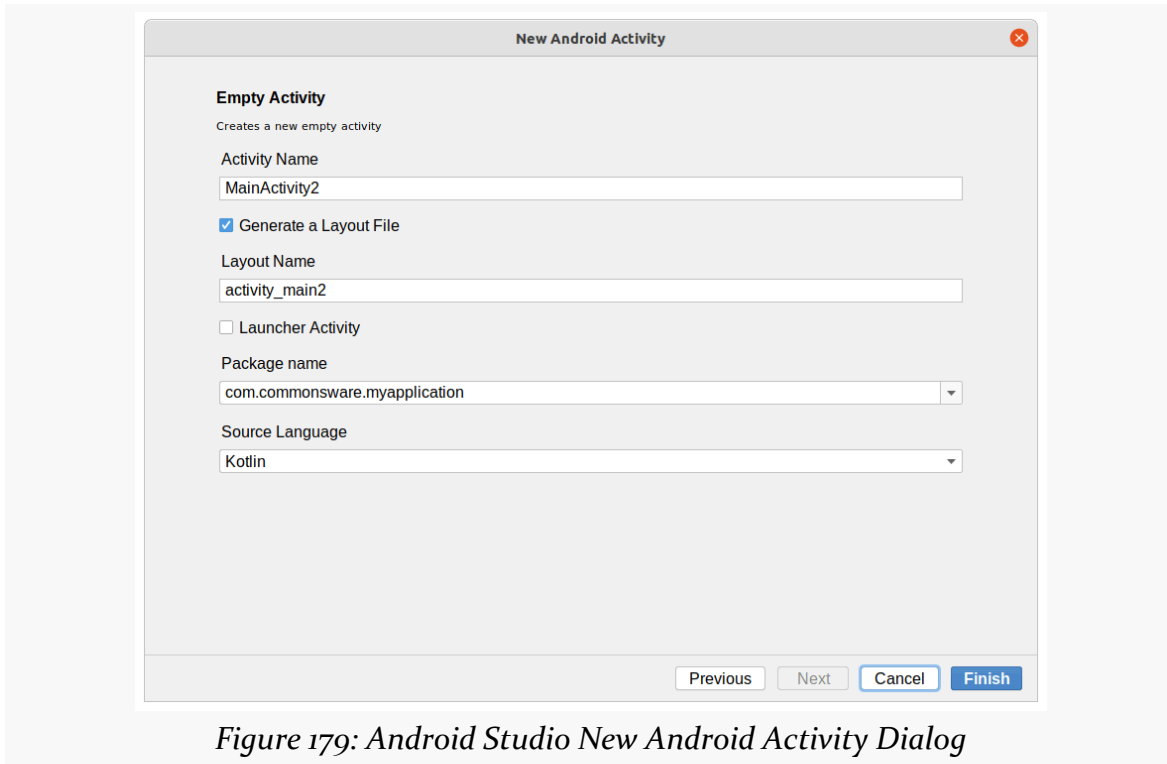


Figure 179: Android Studio New Android Activity Dialog

What you see here will be based upon the template you chose. This happens to be the wizard screen for the “Empty Activity” template; other templates will have forms with other data to collect.

Clicking “Finish” will then create the activity’s Java or Kotlin 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()` function 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).

Augmenting the Manifest

Simply having an activity implementation is not enough. We also need to add it to our `AndroidManifest.xml` file. If you used the new-activity wizard, this entry will be added for you. 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:

```
<activity android:name=".BigSwatchActivity"></activity>
```

(from [TwoActivities/src/main/AndroidManifest.xml](#))

You need the `android:name` attribute at minimum, identifying the Java/Kotlin class that is the implementation of the activity.

As we have seen previously, `<activity>` elements can be more complex:

```
<activity android:name=".MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />

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

(from [TwoActivities/src/main/AndroidManifest.xml](#))

We will explore that `<intent-filter>` a bit more [later in this chapter](#).

Starting Your Own Activity

To start an activity, we call `startActivity()` on some `Context`, typically on our `Activity` or a `Context` obtained from a `View`.

To identify what activity to start, we pass an `Intent` object to `startActivity()`. When starting an activity from your own project, the particular type of `Intent` that we will use is an “explicit” `Intent`. This is where we identify the specific class that implements the activity that we want to start.

The explicit form of the `Intent` constructor takes two parameters:

- A Context, typically the activity that is asking to start another activity
- A Java Class object

That second parameter type means that in Kotlin, you will use `::class.java`, rather than just `::class`, to get to the proper object.

Extra! Extra!

Sometimes, we may wish to pass some data from one activity to the next. For example, we might have a RecyclerView in one activity showing a collection of stuff, and we might have a separate activity to show details of one of those items in the collection. We want to start the detail activity when the user clicks on an item in the RecyclerView. However, somehow, the detail activity needs to know *which* item is the one for which it is to show the details. Unless we tell it which one the user clicked, the detail activity has no way to know.

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. The keys are strings. While you cannot use arbitrary objects for the values, most primitive data types are supported, as are strings and some types of lists. Also, anything implementing Parcelable 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.

Seeing This In Action

As the name suggests, the TwoActivities sample module in the [Sampler](#) and [SamplerJ](#) projects has two activities. The MainActivity is very similar to the Toolbar one from [the previous chapter](#). However, now when the user clicks on a color, we want to launch a second activity that shows that color in a larger form.

The Second Activity

The second activity is named `BigSwatchActivity`, and it was created using the “Empty Activity” template in the Android Studio new-activity wizard. As a result, the

IMPLEMENTING MULTIPLE ACTIVITIES

wizard added a manifest entry for our activity... the same one shown previously in this chapter:

```
<activity android:name=".BigSwatchActivity"></activity>
```

(from [TwoActivities/src/main/AndroidManifest.xml](#))

The new-activity wizard created an `activity_big_swatch` layout resource, which we modified to have a `Toolbar` at the top and a `View` taking up all the remaining space:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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=".BigSwatchActivity">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:background="?attr/colorPrimary"
        android:theme="?attr/actionBarPopupTheme"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />

    <View
        android:id="@+id/swatch"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@id/toolbar" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [TwoActivities/src/main/res/layout/activity_big_swatch.xml](#))

And, the new-activity wizard created the `BigSwatchActivity` class itself, which we then augmented with actual app logic. That class could be written in Java:

IMPLEMENTING MULTIPLE ACTIVITIES

```
package com.commonware.jetpack.samplerj.activities;

import android.os.Bundle;
import com.commonware.jetpack.samplerj.activities.databinding.ActivityBigSwatchBinding;
import androidx.appcompat.app.AppCompatActivity;

public class BigSwatchActivity extends AppCompatActivity {
    static final String EXTRA_COLOR = "color";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ActivityBigSwatchBinding binding =
            ActivityBigSwatchBinding.inflate(getLayoutInflater());

        setContentView(binding.getRoot());

        int color = getIntent().getIntExtra(EXTRA_COLOR, 0x7FFF0000);

        binding.toolbar.setTitle("#" + Integer.toHexString(color));
        binding.swatch.setBackgroundColor(color);
    }
}
```

(from [TwoActivities/src/main/java/com/commonware/jetpack/samplerj/activities/BigSwatchActivity.java](#))

...or Kotlin:

```
package com.commonware.jetpack.sampler.activities

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.commonware.jetpack.sampler.activities.databinding.ActivityBigSwatchBinding

const val EXTRA_COLOR = "color"

class BigSwatchActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val color = intent.getIntExtra(EXTRA_COLOR, 0x7FFF0000)

        ActivityBigSwatchBinding.inflate(layoutInflater).apply {
            setContentView(root)
            toolbar.title = "#{Integer.toHexString(color)}"
            swatch.setBackgroundColor(color)
        }
    }
}
```

(from [TwoActivities/src/main/java/com/commonware/jetpack/sampler/activities/BigSwatchActivity.kt](#))

First, we call `setContentView(R.layout.activity_big_swatch)` to load up our layout resource.

Then, we need to know what color to display. This activity expects to receive that color in the form of an Intent extra. We have an `EXTRA_COLOR` constant defined to use as the key. We call `getIntent()` to retrieve the Intent that created the activity, and on that we call `getIntExtra()` to retrieve the `EXTRA_COLOR` value. The `get...Extra()` functions that return primitives, like an `Int`, take two parameters: the key to use to find the extra, and the default value to return if the extra is not found. In our case, we use a hard-coded gray value as the default.

Then, we:

- Retrieve the Toolbar widget from the layout
- Set the title in the Toolbar to be the color, with a # at the front
- Set the background color of the swatch widget, so our activity shows up almost entirely in that color (other than the Toolbar)

In the case of the Kotlin code, we take advantage of the fact that we only need the binding inside of `onCreate()` to use the `apply()` scope function and skip any sort of property declaration.

Starting the Activity

However, this activity will never be shown to the user unless we call `startActivity()` at some point to show it, ideally passing the desired color as the `EXTRA_COLOR` extra.

The Toolbar sample app — building on previous ones — handles row clicks in `ColorViewHolder`. So, this app just changes `ColorViewHolder` to start `BigSwatchActivity` instead of showing a Toast:

```
package com.commonware.jetpack.samplerj.activities;

import android.content.Context;
import android.content.Intent;
import android.view.View;
import com.commonware.jetpack.samplerj.activities.databinding.RowBinding;
import androidx.recyclerview.widget.RecyclerView;

class ColorViewHolder extends RecyclerView.ViewHolder {
    private final RowBinding row;
    private int color;

    ColorViewHolder(RowBinding row) {
        super(row.getRoot());
    }
}
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
this.row = row;
row.getRoot().setOnClickListener(this::showBigSwatch);
}

void bindTo(Integer color) {
    this.color = color;

    row.label.setText(
        row.label.getContext().getString(R.string.label_template, color));
    row.swatch.setBackgroundColor(color);
}

private void showBigSwatch(View v) {
    Context context = v.getContext();

    context.startActivity(new Intent(context, BigSwatchActivity.class)
        .putExtra(BigSwatchActivity.EXTRA_COLOR, color));
}
}
```

(from [TwoActivities/src/main/java/com/commonsware/jetpack/samplerj/activities/ColorViewHolder.java](#))

```
package com.commonsware.jetpack.sampler.activities

import android.content.Intent
import android.view.View
import androidx.recyclerview.widget.RecyclerView
import com.commonsware.jetpack.sampler.activities.databinding.RowBinding

class ColorViewHolder(private val row: RowBinding) :
    RecyclerView.ViewHolder(row.root) {
    private var color: Int = 0x7FFFFFFF

    init {
        row.root.setOnClickListener(this::showBigSwatch)
    }

    fun bindTo(color: Int) {
        this.color = color
        row.label.text = row.label.context.getString(R.string.label_template, color)
        row.swatch.setBackgroundColor(color)
    }

    private fun showBigSwatch(v: View) {
        val context = v.context

        context.startActivity(
            Intent(context, BigSwatchActivity::class.java)
                .putExtra(EXTRA_COLOR, color)
        )
    }
}
```


IMPLEMENTING MULTIPLE ACTIVITIES

(from [TwoActivities/src/main/java/com/commonsware/jetpack/sampler/activities/ColorViewHolder.kt](#))

We use a method reference to tie our row clicks to a `showBigSwatch()` method. There, we:

- Get a Context from the row View
- Create an explicit Intent, identifying `BigSwatchActivity`
- Add our `EXTRA_COLOR` extra to that Intent with the user's chosen color
- Call `startActivity()` on the Context, passing in our Intent

Now, when the user taps on a color in the list, we see the `BigSwatchActivity`, showing a very large color swatch that probably does not match our Toolbar color very well:

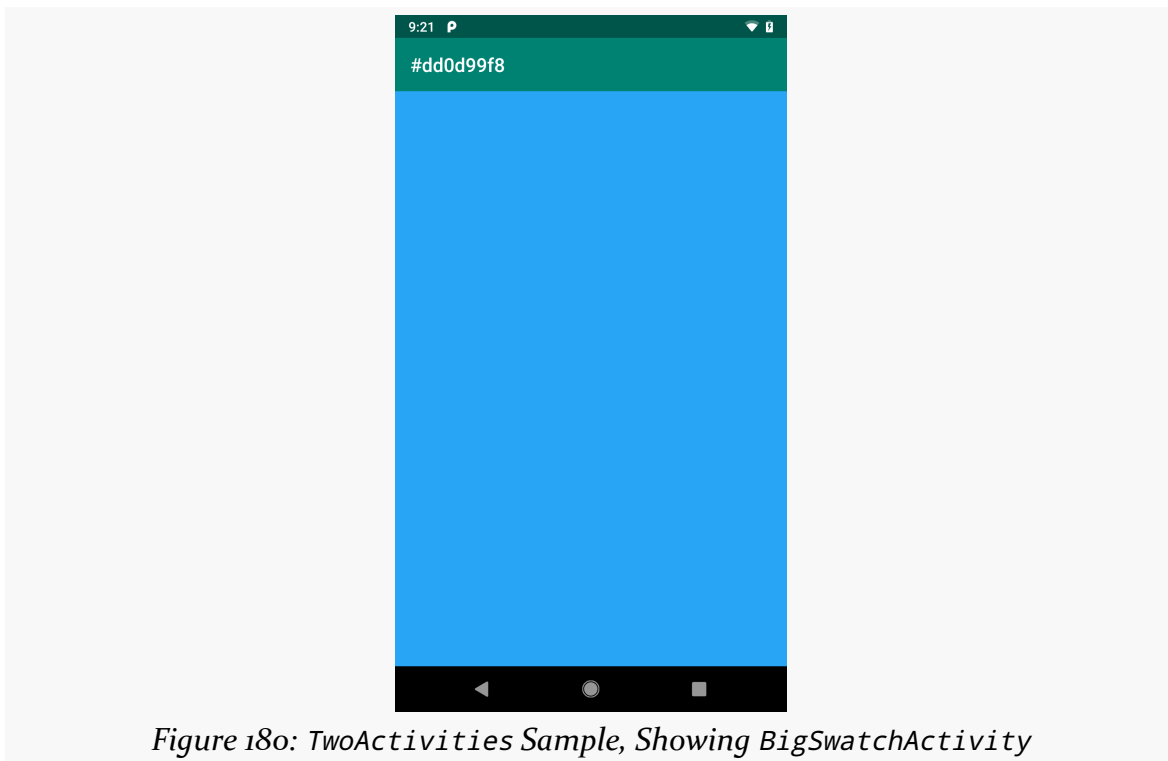


Figure 180: TwoActivities Sample, Showing BigSwatchActivity

Using Implicit Intents

The explicit Intent approach works fine when the activity to be started is one of yours. If you are going to start an activity from some other app, such as a Web browser to view a URL, an explicit Intent will be a problem. After all, you have no idea what Web browser will handle the request (Chrome? Firefox? Brave? Dolphin?

something else?). Plus, you did not write the Web browser and do not know what classes are in it. And, even if you used tools to peek inside the other app and find out its class structure, the developers of that other app could change their implementation at any point.

Instead, you will use what are referred as the “implicit” Intent structure, which looks a 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.

An implicit Intent is made up of two key pieces:

- A `Uri` object indicating what we want to act upon, such as a `Uri` representation of a Web site URL, and
- An action string, identifying the particular action that we want

There are hundreds of action strings that are part of the Android framework, such as:

- `ACTION_VIEW`, to bring up something that can view whatever the `Uri` refers to
- `ACTION_PICK`, to pick something from a collection of somethings
- `ACTION_GET_CONTENT`, to pick something based on a MIME type (e.g., pick an image)
- `ACTION_SEND`, to share some text or content with another app, often used for sending an SMS
- And so on

For example, to try to view a Web page, you can use:

```
startActivity(Intent(Intent.ACTION_VIEW, Uri.parse("https://commonsware.com")))
```

We will see a few other Intent actions from the Android framework over the course

of the rest of the book, starting with the next section. And, it is possible for apps to define their own custom actions — in that case, if the developers of those apps want you using those actions, they will need to document what those actions are and what they do.

Activities and Results

`startActivity()` is asynchronous. The other activity will not show up until sometime later, particularly after you return from whatever callback you were in when you called `startActivity()` (e.g., `onClick()` of some `View.OnClickListener`).

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. Nowadays, though, Google prefers that we wrap these calls up using `ActivityResultContracts`, particularly for SDK-supplied “contracts” for common operations, such as picking a contact.

The `ContactPicker` sample module in the [Sampler](#) and [SamplerJ](#) projects demonstrates `ActivityResultContracts`.

The Scenario

The `MainActivity` UI is pretty simple: two really big buttons, one labeled “Pick” and one labeled “View”. The business rules are:

- The “View” button should be disabled initially
- The “Pick” button, when clicked, should allow the user to pick a contact from the list of contacts on the device
- Once the user picks a contact, the “View” button should be enabled
- The “View” button, when clicked, should show the user details of that particular contact

There are two ways of going about implementing the pick-a-contact and view-a-contact logic:

1. Write a UI ourselves. This is rather complex. In addition, it would require our app to have access to personally-identifying information (PII) about the user's contacts. That requires us to [ask for permission](#), and the user might not want to grant us permission, as letting us scan through their contacts may seem scary.
2. We ask some other app — one that already knows how to work with contacts — to let the user pick a contact and view a contact. On many devices, there will be a built-in “Contacts” app that can do those things on our behalf.

ContactPicker takes the second approach.

The Layout

We have two editions of the activity_main layout resource. One is in the traditional res/layout/ directory, and it has two buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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:padding="@dimen/container_padding">

    <Button
        android:id="@+id/pick"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:text="@string/pick_caption"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHeight_percent="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:enabled="false"
        android:text="@string/view_caption"
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintHeight_percent="0.5"
app:layout_constraintStart_toStartOf="parent" />
```

```
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ContactPicker/src/main/res/layout/activity_main.xml](#))

The other is in `res/layout-w640dp/`, and it has two buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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:padding="@dimen/container_padding">
```

```
    <Button
        android:id="@+id/pick"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:text="@string/pick_caption"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintWidth_percent="0.5" />
```

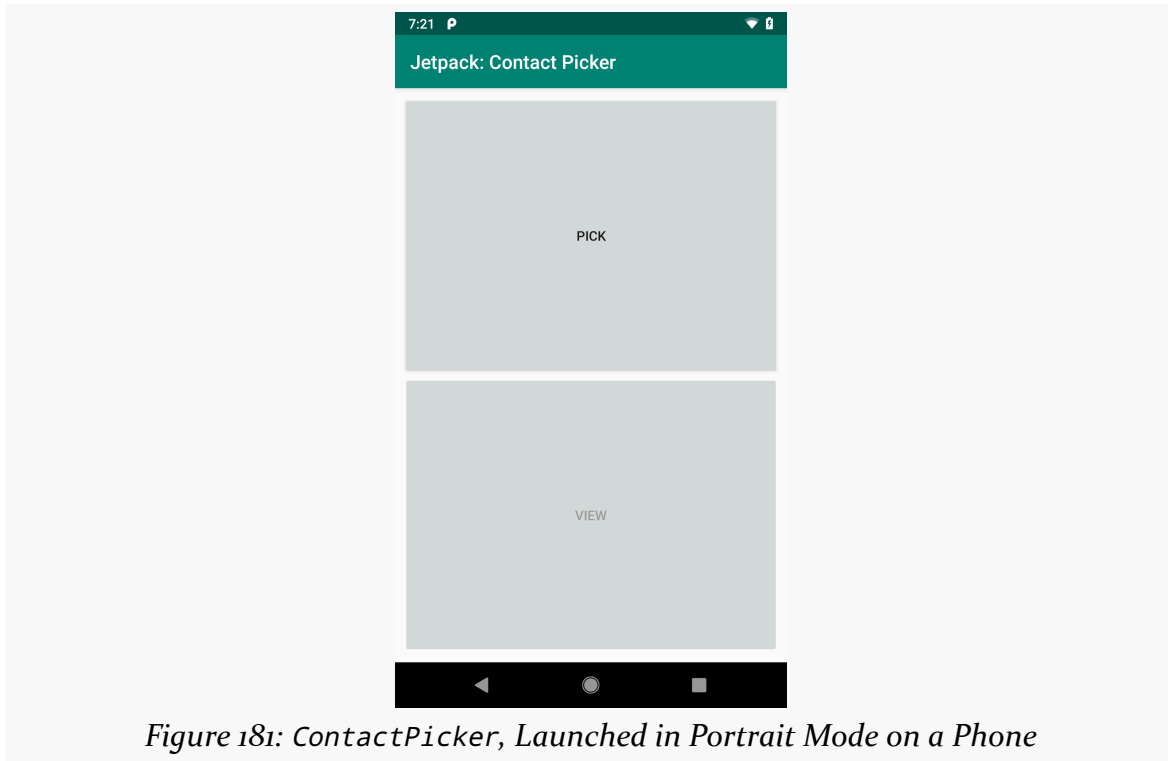
```
    <Button
        android:id="@+id/view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:enabled="false"
        android:text="@string/view_caption"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintWidth_percent="0.5" />
```

```
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ContactPicker/src/main/res/layout-w640dp/activity_main.xml](#))

IMPLEMENTING MULTIPLE ACTIVITIES

The difference is subtle, but the way we have the constraints set up, the `res/layout/` edition of the layout has the two buttons be vertically stacked:



But, on devices with screens wider than 4", the `res/layout-w640dp/` edition of the layout will be used, and it has the constraints set up for the buttons to be side-by-side:

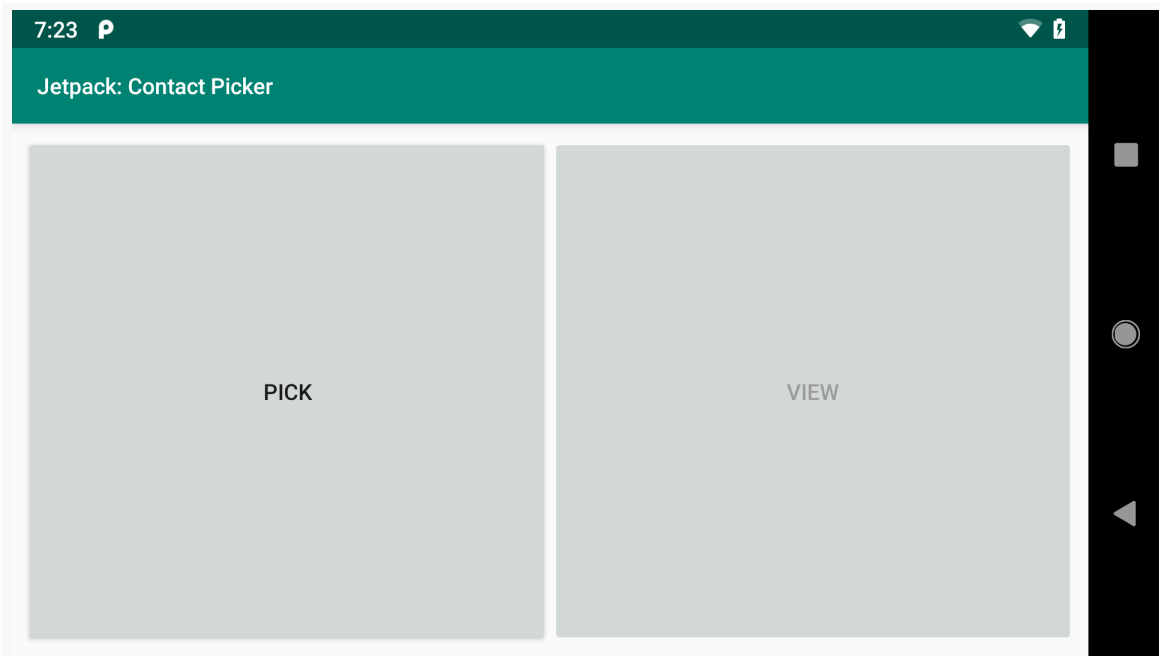


Figure 182: ContactPicker, Launched in Landscape Mode on a Phone

Writing a Contract

The classic way to pick a contact was to use an `ACTION_PICK` implicit Intent with `startActivityForResult()`. The current pattern is to use `registerForActivityResult()` and `ActivityResultContracts.PickContact`:

```
private final ActivityResultLauncher<Void> pickContact =
    registerForActivityResult(new ActivityResultContracts.PickContact(),
        new ActivityResultCallback<Uri>() {
            @Override
            public void onActivityResult(Uri uri) {
                vm.setContact(uri);
                updateViewButton();
            }
        });
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

IMPLEMENTING MULTIPLE ACTIVITIES

```
private val pickContact =  
    registerForActivityResult(ActivityResultContracts.PickContact()) {  
        vm.contact = it  
        updateViewButton()  
    }  
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/MainActivity.kt](#))

`registerForActivityResult()` takes two parameters: an `ActivityResultContract` representing some `startActivityResult()` call that should be made, and an `ActivityResultCallback` where we get the results returned by that other activity. The Jetpack includes several “ready-made” implementations of `ActivityResultContract`, mostly in `ActivityResultContracts`, such as `ActivityResultContracts.PickContact` for picking a contact.

Each `ActivityResultContract` stipulates what it needs for input and what it will deliver as the return value. For `ActivityResultContracts.PickContact`, it needs no input, and it will return a `Uri` object representing the selected contact. The callback will receive a `Uri`, and we need to do something with it. We will explore exactly what we are doing in the callback in an upcoming section.

Note that `registerForActivityResult()` does *not* actually make the request to allow the user to pick a contact. Rather, it sets up an object that we can use later to make that request, and it sets up what we will do with the results (via the callback).

Picking a Contact

In `MainActivity`, we set up a `View.OnClickListener` for the pick button, to allow the user to pick a contact. There, all we do is call `launch()` on our `pickContact` object:

```
binding.pick.setOnClickListener(v -> pickContact.launch(null));
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/MainActivity.java](#))

```
binding.pick.setOnClickListener { pickContact.launch(null) }
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/MainActivity.kt](#))

`launch()` says “go do whatever it is that we configured you to do”. In some cases, we will pass parameters to `launch()` to configure the request — we will see examples of this later in the book. `ActivityResultContracts.PickContact` does not require input, so we pass `null` to `launch()`.

Under the covers, `launch()` will trigger a call to `startActivityForResult()`. In this case, it will start an activity, from the user's contacts app, to allow the user to pick a contact from their list of available contacts.

Getting and Retaining the Contact

The user could use system BACK navigation to exit out of that pick-a-contact activity. In that case, nothing happens in our app.

If, on the other hand, the user chooses a contact, our callback will get control to react to that result. And, in this case, the callback will receive a `Uri` representing the chosen contact. If we got a contact `Uri`, we do two things.

First, we update a `ContactViewModel` that is serving as the viewmodel for this activity:

```
package com.commonware.jetpack.samplerj.contact;

import android.net.Uri;
import androidx.lifecycle.SavedStateHandle;
import androidx.lifecycle.ViewModel;

public class ContactViewModel extends ViewModel {
    private static final String STATE_CONTACT = "contact";
    private final SavedStateHandle state;
    private Uri contact;

    public ContactViewModel(SavedStateHandle state) {
        this.state = state;
        contact = state.get(STATE_CONTACT);
    }

    Uri getContact() {
        return contact;
    }

    void setContact(Uri contact) {
        this.contact = contact;
        state.set(STATE_CONTACT, contact);
    }
}
```

(from [ContactPicker/src/main/java/com/commonware/jetpack/samplerj/contact/ContactViewModel.java](#))

IMPLEMENTING MULTIPLE ACTIVITIES

```
package com.commonware.jetpack.sampler.contact

import android.net.Uri
import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel

private const val STATE_CONTACT = "contact"

class ContactViewModel(private val state: SavedStateHandle) : ViewModel() {
    var contact: Uri? = state[STATE_CONTACT]
    set(value) {
        field = value
        state.set(STATE_CONTACT, value)
    }
}
```

(from [ContactPicker/src/main/java/com/commonware/jetpack/sampler/contact/ContactViewModel.kt](#))

We also call an `updateViewButton()` function that marks the view button as being enabled if we happen to have a contact now:

```
private void updateViewButton() {
    if (vm.getContact() != null) {
        binding.view.setEnabled(true);
    }
}
```

(from [ContactPicker/src/main/java/com/commonware/jetpack/samplerj/contact/MainActivity.java](#))

```
private fun updateViewButton() {
    if (vm.contact != null) {
        binding.view.isEnabled = true
    }
}
```

(from [ContactPicker/src/main/java/com/commonware/jetpack/sampler/contact/MainActivity.kt](#))

In addition, we call `updateViewButton()` in `onCreate()`, after getting our `ContactViewModel`, so we update the view button to be enabled after a configuration change, if appropriate:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
vm = new ViewModelProvider(this).get(ContactViewModel.class);

updateViewButton();
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    updateViewButton()
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/MainActivity.kt](#))

So, the net is that once the user picks the contact, the view button becomes enabled, and we hold onto the contact Uri across configuration changes, so we do not lose track of who the user picked.

Viewing the Contact

Our View.OnClickListener for the view button wraps our contact Uri in an ACTION_VIEW Intent and tries to start an activity to view that contact:

```
binding.view.setOnClickListener(
    v -> {
        try {
            startActivity(new Intent(Intent.ACTION_VIEW, vm.getContact()));
        }
        catch (Exception e) {
            Toast.makeText(this, R.string.msg_view_error,
                Toast.LENGTH_LONG).show();
        }
    });
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

```
binding.view.setOnClickListener {
    try {
        startActivity(Intent(Intent.ACTION_VIEW, vm.contact))
    } catch (e: Exception) {
        Toast.makeText(this, R.string.msg_view_error, Toast.LENGTH_LONG).show()
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/MainActivity.kt](#))

It is theoretically possible that there is no activity to allow the user to view one of their contacts. In practice, that should not be a problem — if the user can *pick* a contact, they almost certainly can *view* a contact. However, to be safe, we wrap our `startActivity()` call in a try/catch block to deal with any possible exceptions. In particular, if there is no activity to handle a particular `startActivity()` (or `startActivityForResult()`) call, Android will throw an `ActivityNotFoundException`, which we would catch.

If you run this on a device or emulator that has a working “contacts” app with 1+ contacts in it, clicking the “Pick” button lets you pick a contact, and then clicking the “View” button views details of the picked contact. In both cases, the contacts app is supplying the UI for picking and viewing contacts.

The Inverse: `<intent-filter>`

When we use an explicit Intent with `startActivity()`, it is fairly clear how Android determines what activity to display: it is the one whose class is listed in the Intent.

When we use an implicit Intent — one with an action string and maybe a `Uri`, but no class — somehow Android needs to find out what activity (or activities) can handle that.

This is where the `<intent-filter>` comes in.

In our manifest, inside of an `<activity>` element, you can optionally have one or more `<intent-filter>` elements. These describe certain implicit Intent structures that our activity can handle. If some code tries starting an implicit Intent with a matching structure, our activity will be considered a candidate. So, in the case of our `ACTION_PICK` and `ACTION_VIEW` requests in the preceding sections, we are hoping that there are 1+ activities with an `<intent-filter>` that matches our Intent.

Most apps will have an activity with one particular `<intent-filter>`:

```
<intent-filter>
  <action android:name="android.intent.action.MAIN" />

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

(from [ContactPicker/src/main/AndroidManifest.xml](#))

This would match an Intent like:

```
Intent(Intent.ACTION_MAIN).addCategory(Intent.CATEGORY_LAUNCHER)
```

A “category” is simply an identifier of related Intent structures and is one facet of an implicit Intent, along with things like the action string and “data” Uri.

A launcher can ask Android “what activities support this Intent?”. Android will return a list of matches, and the launcher can use this to provide options for the user. When the user clicks on a *specific* activity, then the launcher has the details to create an explicit Intent, which it can use to start the activity. Since this is the standard behavior of launchers, and since most Android apps want to have 1+ activities appear in the launcher, most apps will have 1+ activities with the MAIN/LAUNCHER <intent-filter>.

Most apps will not need <intent-filter> on an <activity> beyond this one, though. However, they might *use* other apps that have activities with other <intent-filter> options, such as the ACTION_PICK and ACTION_VIEW scenarios.

We will see an example of this sort of structure in [an upcoming chapter](#), where we will examine an app that supports ACTION_SEND, the standard Intent action for “share” options in apps.

Adding Library Modules

When you create a new Android Studio project, by default you get a single module named `app`. For many simple projects, that is all that you need.

The `Sampler` and `SamplerJ` projects have lots of modules. All but one of these are application modules, ones that create Android apps. Ordinarily, each of these apps might be in a separate project as an app module — they are combined into a single project for your convenience:

- You only have to check out a single Git repository
- You can easily flip from one sample to another in a single IDE window

However, one module in the `Sampler Kotlin` project, named `Utilities`, is unlike the other modules. `Utilities` is a library module, not an application module. A library module does not build an Android app. Instead, it is just a container for code, resources, and assets that can be used by other modules.

In this chapter, we will explore briefly what the value is of library modules and how to use library modules that somebody created. Later in the book, we will see [how to create library modules of your own](#).

Reasons for Library Modules

There are lots of reasons why you might want to have one or more library modules in a project. Here are just a few of them.

Reuse Across App Modules

The reason for the `Utilities` library module in this project is to reuse code across

multiple other modules. You can define classes once in the library and reuse them in the other modules, rather than having to maintain a bunch of parallel copies of that class in those other modules.

Utilities contains a single class — `ViewBindingFragment` — that we will use in several sample apps over the course of the rest of the book. We will see what `ViewBindingFragment` is in [the next chapter](#).

It is also possible to publish a library in a way that it can be used by apps implemented in other projects. Each of the `implementation` lines that we see in our module's `build.gradle` file represents some library that somebody else wrote and published.

Reduce Build Time

Suppose we have an app module that depends on 7 library modules. The app module represents 30% of the total code of the project, and each of the library modules contributes an additional 10%. In this case:

- Changing code in app usually only requires building things in the app module itself, not the libraries
- Changing code in a library would require building the library, plus building any modules that depend upon that library, such as the app module

Making a change to the app, therefore, might require building 30-40% of the code. If all of the code were in the app module, with no libraries, making a change to the app would require building 100% of the code.

In reality, the story here is a *lot* more complex, as build times depend on what you are changing (Java/Kotlin code? resources? manifest entries?) as well as the module organization. Smaller projects will not gain much build speed from this sort of module separation. Large projects, though, might build a lot faster, enough to make the headache of organizing the code this way be worthwhile.

Define Internal APIs

Normally, in Kotlin, everything is `public` unless you expressly declare it to be `private`. In a single-module project, those are your only two visibility options.

However, in a multi-module project, you can take advantage of Kotlin's internal visibility modifier. `internal` says:

- Treat it as `public` with respect to other code in this module
- Treat it as `private` with respect to code in other modules that depend upon this module

Hence, you can use `internal` to better define the “contracts” between portions of code, to better delineate what is the “API surface” that others can use and what is internal implementation that should not be used. When everything resides in a single module, `internal` works the same as `public` and has no value.

Consuming a Library Module

If somebody else has created a library module in your project, you can depend upon it from some other module using the same sort of implementation statement that we use to depend upon Google-supplied and third-party libraries. The difference is that rather than using a string like

`androidx.constraintlayout:constraintlayout:2.0.4` to identify the library, you use `project(":...")`, where `...` represents the name of the module.

So, for example, an app module that wants to use the Utilities module would have:

```
coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
```

(from [FragmentManual/build.gradle](#))

Adopting Fragments

Activities are fine, but they are fairly inflexible. Development is moving away from using activities as the core foundation of our UI. Activities will always exist, but many times we use an activity mostly as a simple container for other UI logic.

And, frequently, that UI logic is held in fragments.

The original vision for fragments was to make it easier to support early Android tablets, allowing you to assemble tablet-sized UIs by snapping together a bunch of phone-sized UIs that you use individually on phones.

Over time, fragments were used in more places and for more reasons. While fragments are not required, Google strongly encourages their use. Jetpack specifically advocates having an app be a single activity, with fragments for each “screen” of information.

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 How](#). 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, at that point, you can use fragments in different ways, with two main patterns being:

- Having one activity switch between fragments based on user input
- Having multiple fragments on the screen at once, perhaps to take better advantage of larger screen sizes on tablets, Chromebooks, etc.

Functionally, fragments are Java/Kotlin classes, extending from a base Fragment class.

Where??

Fragments will appear on the screen where you tell them to appear. There are two main approaches for this:

- Having <fragment> elements in layout resources
- Having FragmentContainerView elements in layout resources, where you then supply the fragments to put in those containers at runtime

The first approach is for cases where the fragment will always be shown (“static fragments”) by the activity. The second approach is for cases where the fragment might be swapped out for another fragment, or whether the fragment might be conditionally shown based on user input, screen size, or other criteria (“dynamic fragments”).

Of the two, dynamic fragments are *much* more common.

Who?!?

Many fragments you will write yourself, just as you write your activities and other application code.

However, libraries can also contribute fragment implementations. We will see an example of this [in the next chapter](#), with a Google-supplied Jetpack library offering a fragment that we use directly. Third party libraries might offer fragments as part of

their API, for you to use or subclass as appropriate.

When?!?!?

In modern, Jetpack-centric app development, often we define our fragments at the outset, as part of building our UI.

It is certainly possible to take an app that has several activities and rewrite it such that it has one activity and several fragments. This is a bit tedious, but it might prove necessary at some point for legacy projects.

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?

There are many reasons for using fragments, some of which were hinted at above.

Break the Intent Barrier

Sharing data between activities is a problem. We can use extras on Intents, but they are limited in terms of size and data types. Other than that, we are limited to using singletons or other global objects, and those run the risk of memory leaks and related problems.

However, a single activity can have several fragments and switch between them at will. Plus, an activity can work directly with its fragments and share things, such as having a shared `ViewModel`. So, fragments give us ways of having different “screens” while still allowing everything to work using more-or-less normal Java/Kotlin object interactions.

To an extent, you can draw a parallel to Web development. Early Web development, and even a lot of modern development, has each “screen” be a separate Web page at a separate URL, fetched from the server. This works, but it has a similar problem to multiple activities in Android: one Web page cannot directly work with the DOM or JavaScript objects of another Web page. In the past decade, there has been a lot of movement towards “single-page applications”, where a single URL loading a single Web page has multiple screens’ worth of content, courtesy of lots of DOM rewriting. While this has its own set of problems, it gets past the separate-pages limitations,

allowing multiple screens to share a common set of JavaScript objects.

Decomposition

Technically, one does not need fragments to allow for a single activity to represent multiple screens. We have used `setContentView()` to populate the activity's UI, and you can call that as many times as needed. Each `setContentView()` call replaces whatever the previous “content view” was with a new one. Or, you can do other things with the activity's view hierarchy to change the UI: hide and show widgets, add and remove widgets, etc.

However, if you go this route, you run the risk of having a single monster activity trying to manage all of this logic. For ease of maintenance and testing, having some decomposition is useful, so a set of Java/Kotlin classes can manage an individual screen, while the activity orchestrates which screen is seen at any point in time.

Fragments offer a foundation for this sort of decomposition. It is not the only option, and it may not be the absolute best option. However, it is *Google's* best option, and Jetpack reflects a fragment-centric view of Android app development.

Screen Sizes

The original 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, to monsters that are over 6”. However, those variations in screen size pale in comparison to the differences between phones and tablets, or phones and Chromebooks.

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.

ADOPTING FRAGMENTS

Part of the original vision for fragments was that one could assemble a tablet UI from a collection of phone screens, side by side.

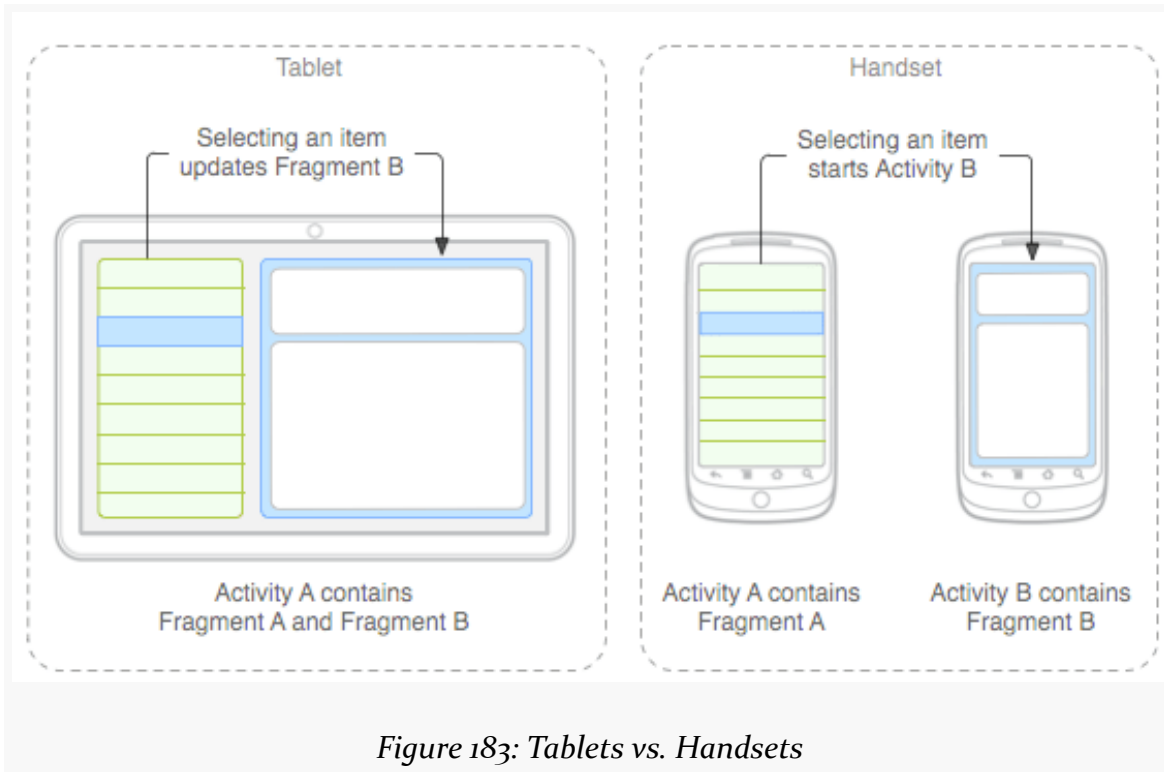


Figure 183: Tablets vs. Handsets

(the above image is 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](#))

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.

OMGOMGOMG, HOW?!?!??

Well, answering that question is what the rest of this chapter is for!

Where You Get Your Fragments From

We use a `Fragment` class as the basis for our fragments. There are multiple `Fragment` class implementations, though, which makes things confusing.

The Jetpack edition of fragments is in the `androidx` namespace, specifically `androidx.fragment.app.Fragment`. That is the `Fragment` class that we will be using in this book.

The Jetpack `Fragment` class is based on the one from the Android Support Library. If you see references to `android.support.v4.app.Fragment`, that is the Android Support Library edition. This is very obsolete and should not be used.

You will also find `android.app.Fragment`. This is the framework implementation of fragments. It has been officially deprecated, with Google steering developers towards a library-based implementation. The problem is that fragments have had quite a few bugs over the years. Framework classes are only updated when the OS is updated, and for many users that means the framework classes are rarely updated and are usually out of date. By contrast, *you* as the app developer control which version of the libraries that you use, so you can ensure that you are using up-to-date versions that contains relevant bug fixes.

Fortunately, Java and Kotlin are strongly-typed languages, so it will be difficult for you to accidentally use the wrong `Fragment`... though “difficult” is not “impossible”.

Static vs. Dynamic Fragments

Sometimes, you will have a fragment that should be around as long as your activity is around. For that, you can use “static fragments”, where you use an element in a layout resource to specify where the fragment should go and how big it should be.

More often, though, your fragments will come and go, based on user input:

- You start by showing a list of stuff
- The user clicks on an item in the list, and you show details about the item that the user clicked on
- The user clicks an “edit” option in the toolbar, and so you show an edit form to allow the user to modify the item
- The user clicks BACK to return to the details, then BACK again to return to the list

In all of these cases, in a fragment-based UI, you will use dynamic fragments.

Roughly speaking, there are two ways of employing dynamic fragments:

1. Manually, using `FragmentManager` and `FragmentTransaction` classes, as we will explore in this chapter
2. Using the Jetpack Navigation component, which we will see in [the next chapter](#)

Fragments, and What You Have Seen Already

Fragments can do most everything of what we have seen so far that activities can do:

- Fragments can use layout resources to show a UI
- Fragments can have a `ViewModel` — in fact, they can not only have their own, but they can share a `ViewModel` with the activity that hosts them
- Fragments have [lifecycle methods](#), including both ones that you see in activities (e.g., `onCreate()` and `onDestroy()`) and ones that are distinct for fragments (e.g., `onAttach()` and `onDetach()`)

ToDo, or Not ToDo? That Is the Question

To see dynamic fragments in action, let's turn to the `FragmentManual` sample module in the [Sampler](#) and [SamplerJ](#) projects.

This sample app is relatively complicated. We have:

- One activity
- Two fragments
- Two viewmodels
- A `RecyclerView`
- A few layout resources
- And other stuff besides

What We're Building

The sample app implements a to-do list, showing a list of to-do items along with details of an individual item.

This code mimics some of the code from [Exploring Android](#), though with a number

ADOPTING FRAGMENTS

of differences, to illustrate the manual use of dynamic fragments.

From the user's standpoint, launching the app brings up a list of to-do items, where the list of things to do is hard-coded in the app:

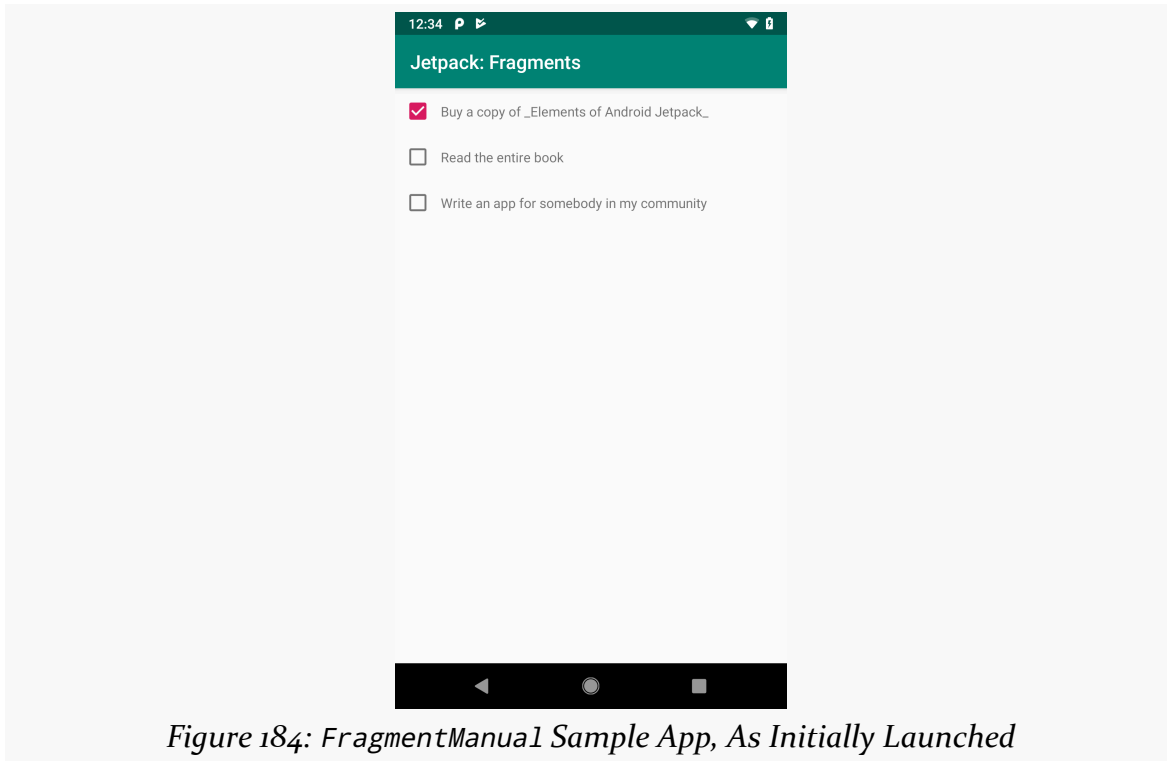


Figure 184: FragmentManual Sample App, As Initially Launched

Tapping on the description of an item brings up additional details, including the creation date and, in some cases, some notes:

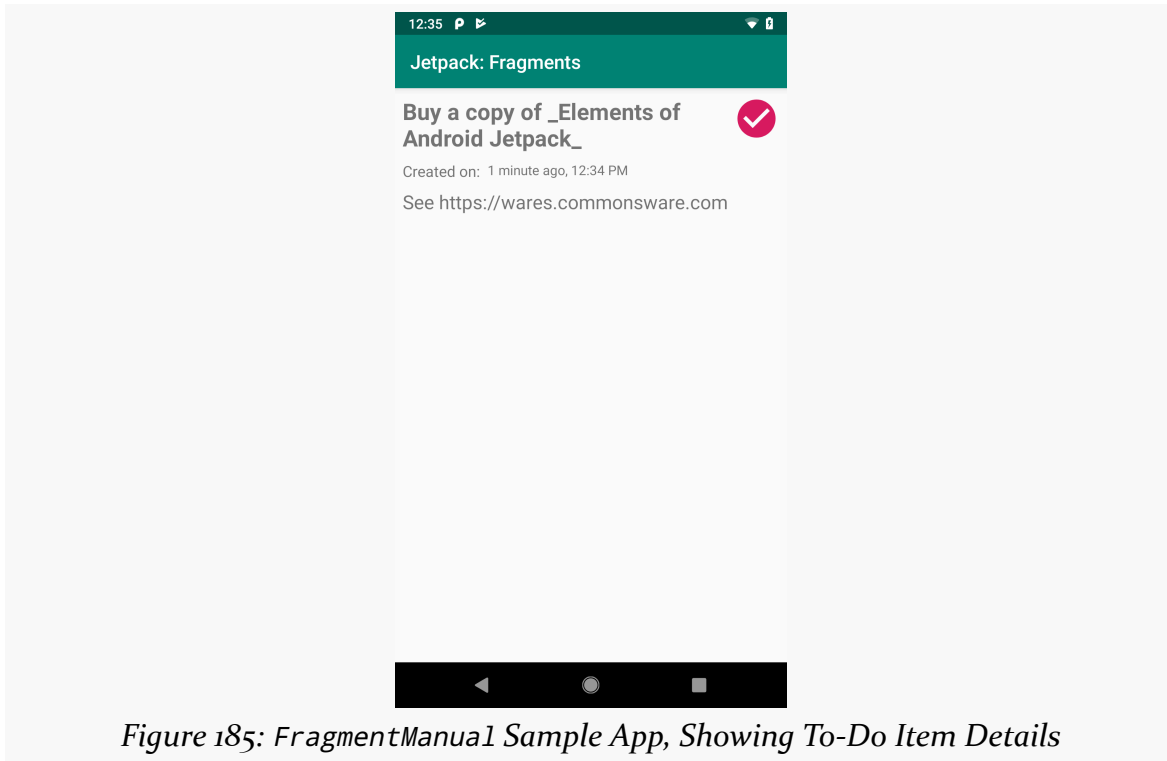


Figure 185: FragmentManual Sample App, Showing To-Do Item Details

This version of the app is read-only, though readers of *Exploring Android* will create a version of this app that allows users to add, edit, and remove to-do items.

The Model

Objects that represent our “business data” are usually called “model objects”. In a to-do list app, we need objects that represent to-do items, and those would be considered model objects.

So the sample app has a `ToDoModel` class that holds onto things like the description (e.g., “Write an app for somebody in my community”), the creation time, and so forth.

There are five pieces of data that we want to track:

- The description, as mentioned above
- Whether or not the item is completed, which will control things like

whether the CheckBox is checked in the list and whether the checkmark icon shows up in the detail screen

- Some optional additional notes about the item (e.g., "Talk to some people at non-profit organizations to see what they need!")
- The time this item was created
- Some sort of unique identifier — while this is not important now, it will be later when [we start persisting this data in a database](#)

Functionally, the Java and Kotlin editions of this class are identical. The code, though, winds up being a fair bit different.

Kotlin

In Kotlin, we can use a simple data class, with individual `val` properties for the five pieces of data mentioned above:

```
data class ToDoModel(  
    val id: String,  
    val description: String,  
    val isCompleted: Boolean = false,  
    val notes: String? = null,  
    val createdOn: Instant = Instant.now()  
)
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoModel.kt](#))

Of particular interest:

- `isCompleted` is defaulted to `false`
- `notes` can be `null` and is defaulted to `null`
- `createdOn` is defaulted to the current date and time

Because this is a data class, we get things like a `copy()` function, `toString()`, `equals()`, and so forth “for free”, courtesy of the Kotlin compiler.



You can learn more about data classes in the "Data Class" chapter of [Elements of Kotlin](#)!

Java

We want the same basic characteristics in Java. It requires a bit more code:

```
public class ToDoModel {
    @NonNull
    public final String id;
    @NonNull
    public final String description;
    public final boolean isCompleted;
    @Nullable
    public final String notes;
    @NonNull
    public final Instant createdOn;

    ToDoModel(@NonNull String id, @NonNull String description,
              boolean isCompleted, @Nullable String notes,
              @NonNull Instant createdOn) {
        this.id = id;
        this.description = description;
        this.isCompleted = isCompleted;
        this.notes = notes;
        this.createdOn = createdOn;
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoModel.java](#))

We have fields for each of the five pieces of data. Those that are objects are marked with `@Nullable` and `@NotNu11` annotations, so that Android Studio can help ensure that we use them properly with respect to null values. The fields are marked as `final`, which is as close as Java comes to the immutability that you get with `val` properties in a Kotlin data class.

We also have a constructor to fill in those fields. However, we do not have default parameter values, so the caller will need to supply all five pieces of data whenever it creates an instance of `ToDoModel`.

A Sidebar About Instant

`ToDoModel` uses `Instant`. `Instant` was added to Java in Java 8, but it did not show up in the Android SDK until API Level 26. On the surface, this would suggest that we should not be using `Instant`, as the `minSdkVersion` of the Java and Kotlin projects is 21. Using newer classes on older devices usually results in a crash, such as a `ClassNotFoundException`.

ADOPTING FRAGMENTS

In this case, though, it works.

The reason is that Google is taking some steps to allow some Java 8 features to be used on older devices. Support for things like Java 8 lambda expressions have been around for a few years. In 2020, they added support for some Java 8-specific types, Instant being one of them.

To be able to support Instant on older devices, you need two items in your module's build.gradle file. The first is a `coreLibraryDesugaringEnabled true` directive in the `compileOptions` closure in the `android` closure:

```
compileOptions {  
    coreLibraryDesugaringEnabled true  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}
```

(from [FragmentManual/build.gradle](#))

The other is a `coreLibraryDesugaring` dependency directive to pull in some version of `com.android.tools:desugar_jdk_libs`, as part of your dependencies:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.recyclerview:recyclerview:1.2.1'  
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.1'  
    implementation project(":Utilities")  
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'  
}
```

(from [FragmentManual/build.gradle](#))

The Repository

Repositories are a common pattern in modern Android app development.

The idea of a repository is to isolate details of how data is stored from the UI that is using that data. The UI should neither know nor care whether the data is stored in a database, in some other type of file, or on a Web service. Similarly, the code that handles the data storage should not care whether the data is represented visually in fields, checkboxes, or other sorts of widgets. Having a clean boundary between “the stuff that stores data” and “the stuff that uses data” can also help with testing. We

will explore the repository pattern in greater detail [later in the book](#).

For now, though, we need *something* that can hold a few fake `ToDoModel` instances, so we may as well set up a `ToDoRepository` for that.

A real repository can be very complicated. This sample app has no actual data storage, so our `ToDoRepository` is fairly trivial:

```
package com.commonware.jetpack.samplerj.fragments;

import java.time.Instant;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;

class ToDoRepository {
    static final ToDoRepository INSTANCE = new ToDoRepository();
    private final Map<String, ToDoModel> items = new HashMap<>();

    private ToDoRepository() {
        add(new ToDoModel(UUID.randomUUID().toString(),
            "Buy a copy of _Elements of Android Jetpack_", true,
            "See https://wares.commonware.com",
            Instant.now()));
        add(
            new ToDoModel(UUID.randomUUID().toString(), "Read the entire book",
                false, null,
                Instant.now()));
        add(new ToDoModel(UUID.randomUUID().toString(),
            "Write an app for somebody in my community", false,
            "Talk to some people at non-profit organizations to see what they need!",
            Instant.now()));
    }

    @NonNull
    List<ToDoModel> getItems() {
        return new ArrayList<>(items.values());
    }

    @Nullable
    ToDoModel findItemById(String id) {
        return items.get(id);
    }
}
```

```
private void add(ToDoModel model) {  
    items.put(model.id, model);  
}  
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoRepository.java](#))

```
package com.commonsware.jetpack.sampler.fragments  
  
import java.util.*  
  
object ToDoRepository {  
    private val items = listOf(  
        ToDoModel(  
            id = UUID.randomUUID().toString(),  
            description = "Buy a copy of _Elements of Android Jetpack_",  
            isCompleted = true,  
            notes = "See https://wares.commonsware.com"  
        ), ToDoModel(  
            id = UUID.randomUUID().toString(),  
            description = "Read the entire book"  
        ), ToDoModel(  
            id = UUID.randomUUID().toString(),  
            description = "Write an app for somebody in my community",  
            notes = "Talk to some people at non-profit organizations to see what they need!"  
        )  
    ).associateBy { it.id }  
  
    fun getItems(): List<ToDoModel> = items.values.toList()  
  
    fun findItemById(id: String) = items[id]  
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoRepository.kt](#))

The repository holds a Map of `ToDoModel` objects, keyed by their `id` value. It has functions to retrieve all models or a single model by ID.

The repository is set up as a singleton, either using a static field in Java or simply having the repository be an object in Kotlin. That works, but it is not very flexible. [Later in the book](#), we will explore ways in which our app can use the repository like a singleton, yet we would have the ability to swap out implementations of the repository to use in different scenarios, through what is known as “dependency inversion”.

The DisplayFragment

We need to display a `ToDoModel` to the user. And, in this sample app, we want to use fragments for that. So, we have a `DisplayFragment` that will fill that role. That `DisplayFragment` has its own layout and viewmodel, just like an activity might. The

Java/Kotlin code for the fragment is a bit different than what you would see in an activity, but not *that* different.

So, let's look at each of the pieces individually.

The Layout

The `todo_display` layout resource represents the UI that we want to use to show a `ToDoModel` to the user:

```
<?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="model"
            type="com.commonware.jetpack.sampler.fragments.ToDoModel" />

        <variable
            name="createdOnFormatted"
            type="java.lang.CharSequence" />

        <import type="android.view.View" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <ImageView
            android:id="@+id/completed"
            android:layout_width="@dimen/checked_icon_size"
            android:layout_height="@dimen/checked_icon_size"
            android:layout_marginEnd="8dp"
            android:layout_marginTop="8dp"
            android:contentDescription="@string/is_completed"
            android:src="@drawable/ic_check_circle_black_24dp"
            android:tint="@color/colorAccent"
            android:visibility="@{model.isCompleted ? View.VISIBLE : View.GONE}"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/desc"
```



```
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_marginEnd="8dp"
android:layout_marginStart="8dp"
android:layout_marginTop="8dp"
android:text="@{model.description}"
android:textSize="@dimen/desc_view_size"
android:textStyle="bold"
app:layout_constraintEnd_toStartOf="@+id/completed"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent" />
```

<TextView

```
android:id="@+id/label_created"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginStart="8dp"
android:layout_marginTop="8dp"
android:text="@string/created_on"
android:textSize="@dimen/created_on_size"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toBottomOf="@+id/desc" />
```

<TextView

```
android:id="@+id/created_on"
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_marginEnd="8dp"
android:layout_marginStart="8dp"
android:layout_marginTop="8dp"
android:text="@{createdOnFormatted}"
app:layout_constraintEnd_toStartOf="@+id/completed"
app:layout_constraintStart_toEndOf="@+id/label_created"
app:layout_constraintTop_toBottomOf="@+id/desc" />
```

<TextView

```
android:id="@+id/notes"
android:layout_width="0dp"
android:layout_height="0dp"
android:layout_marginBottom="8dp"
android:layout_marginEnd="8dp"
android:layout_marginStart="8dp"
android:layout_marginTop="8dp"
android:text="@{model.notes}"
android:textSize="@dimen/notes_size"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent"
```

```
app:layout_constraintTop_toBottomOf="@+id/label_created" />

</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [FragmentManual/src/main/res/layout/todo_display.xml](#))

On the whole, this is a basic `ConstraintLayout`-based layout resource, akin to those that we have seen previously in the book. It uses [data binding](#) to populate the core widgets.

The most interesting widget of the lot, though, is the `ImageView`:

```
<ImageView
    android:id="@+id/completed"
    android:layout_width="@dimen/checked_icon_size"
    android:layout_height="@dimen/checked_icon_size"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="8dp"
    android:contentDescription="@string/is_completed"
    android:src="@drawable/ic_check_circle_black_24dp"
    android:tint="@color/colorAccent"
    android:visibility="@{model.isCompleted ? View.VISIBLE : View.GONE}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

(from [FragmentManual/src/main/res/layout/todo_display.xml](#))

The icon that we use — `ic_check_circle_black_24dp` — is one imported from clip art through the Vector Asset wizard, the same way that we setup Toolbar icons in [an earlier chapter](#). However, we want it to show up in a different color. We could hand-modify the XML of the resource. A simpler approach is used here, where we use `android:tint` to apply our `colorAccent` color to the widget. All black pixels will be replaced with pixels based on our accent color. This way, if we elect to change the accent color, changing the color resource affects this `ImageView` without having to modify the drawable itself.

Also, the `ImageView` uses a slightly complicated data binding expression:

```
android:visibility="@{model.isCompleted ? View.VISIBLE : View.GONE}"
```

(from [FragmentManual/src/main/res/layout/todo_display.xml](#))

We want the icon to be visible if the `isCompleted` property is true. The `android:visibility` attribute handles this, with three possible values:

- `visible`
- `invisible`: the widget's pixels are not drawn, but it still takes up space on the screen
- `gone`: the widget is ignored entirely

We want to toggle the visibility to be `visible` or `gone` based on whether the to-do item has been completed.

Here, we use a “ternary expression”. This mimics the ternary expressions available in Java (though not in Kotlin). A ternary expression is made up of three pieces:

- To the left of the `?` is some boolean value — in our case, the value of the `isCompleted` property
- To the left of the `:` is the value to use if the boolean value is `true`
- To the right of the `:` is the value to use if the boolean value is `false`

The end result is that the user sees a tinted checkmark if the to-do item is completed.

The ViewModel

`DisplayFragment` has a `DisplayViewModel` to hold its data across configuration changes:

```
package com.commonware.jetpack.samplerj.fragments;

import androidx.lifecycle.ViewModel;

public class DisplayViewModel extends ViewModel {
    private ToDoModel model;

    ToDoModel getModel(String id) {
        if (model == null) {
            model = ToDoRepository.INSTANCE.findItemById(id);
        }

        return model;
    }
}
```

(from [FragmentManual/src/main/java/com/commonware/jetpack/samplerj/fragments/DisplayViewModel.java](https://github.com/CommonWare/jetpack-samplerj/blob/master/src/main/java/com/commonware/jetpack/samplerj/fragments/DisplayViewModel.java))

```
package com.commonware.jetpack.sampler.fragments

import androidx.lifecycle.ViewModel

class DisplayViewModel : ViewModel() {
    private var model: TodoModel? = null

    fun getModel(id: String) =
        model ?: TodoRepository.findById(id).also { model = it }
}
```

(from [FragmentManual/src/main/java/com/commonware/jetpack/sampler/fragments/DisplayViewModel.kt](#))

Each instance of our DisplayFragment will display a single TodoModel instance. The fragment starts out with just the ID of the model, though, as we will see in the next section. Our DisplayViewModel just asks our TodoRepository for the TodoModel associated with that ID.

The Class Declaration

As noted earlier in the chapter, we are using `androidx.fragment.app.Fragment` as our Fragment implementation, so DisplayFragment extends from that class.

Fragments should have a public zero-argument constructor. That is because the fragment code — such as the AndroidX fragment code — will create instances of our fragments for us after configuration changes. The only constructor that this code knows how to use is a public zero-argument constructor.

In Java, we skip any constructor and just extend from Fragment:

```
public class DisplayFragment extends Fragment {
```

(from [FragmentManual/src/main/java/com/commonware/jetpack/samplerj/fragments/DisplayFragment.java](#))

We will look at the Kotlin edition of the fragment later in the chapter.

The Factory Function

Frequently, our fragments need to know what they are supposed to do. In the case of DisplayFragment, it needs to know what TodoModel to display. Given DisplayViewModel, we can get a TodoModel given its ID... but we still need for DisplayFragment to get that ID. Since we are using zero-argument constructors, we need another way to get that ID over to the fragment.

With manual fragments, that typically involves a factory function that can create instances of our `DisplayFragment` for us.

In Java, that would be a static method that returns a `DisplayFragment`:

```
static DisplayFragment newInstance(String modelId) {  
    DisplayFragment result = new DisplayFragment();  
    Bundle args = new Bundle();  
  
    args.putString(ARG_MODEL_ID, modelId);  
    result.setArguments(args);  
  
    return result;  
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

The Arguments Bundle

These functions take a `modelId` as input and return a `DisplayFragment` as output. What happens in between is that the function attaches an “arguments Bundle” to the fragment, putting that model ID String into the Bundle.

We do this to handle the case where our process is terminated while in the background, but the user returns to our activity and its fragments quickly. The arguments Bundle for the fragment forms part of the saved instance state of our activity. This way, we can hold onto the model ID and be able to show that model again... in theory.

In reality, we generate fresh model objects with fresh IDs whenever we get a fresh process, so the ID from the old process will be wrong. That is because this example is very fake and does not persist its model data in a database. We will correct that limitation [in a later chapter](#).

We use a constant named `ARG_MODEL_ID` as the key to the value for the Bundle:

```
private static final String ARG_MODEL_ID = "modelId";
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

In Java, we create that Bundle like we would any other object.

So, overall, the `newInstance()` factory function:

- Creates an instance of `DisplayFragment`
- Creates a `Bundle`
- Puts the model ID into the `Bundle` under the `ARG_MODEL_ID` key
- Attaches the `Bundle` to the fragment via `setArguments()`
- Returns the `DisplayFragment` with the attached `Bundle`

The `onCreateView()` Function

A fragment typically has an `onCreateView()` function, whether on its own or one that it inherits from some superclass. The job of `onCreateView()` is to return the `View` that represents the view hierarchy to be managed by this fragment. In the case of `DisplayFragment`, we want to use the view hierarchy defined by the `todo_display` layout resource.

We are using data binding in this sample, so our `todo_display` layout resource gives us a `TodoDisplayBinding` class, courtesy of the data binding code generator. So, as we did in previous data binding examples, we can use that binding class to `inflate()` our UI and return the `getRoot()` `View` from `onCreateView()`:

```
@Nullable
@Override
public View onCreateView(@NonNull LayoutInflater inflater,
                        @Nullable ViewGroup container,
                        @Nullable Bundle savedInstanceState) {
    binding = TodoDisplayBinding.inflate(inflater, container, false);

    return binding.getRoot();
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

We are also holding onto the binding in a binding property for later use.

The `onViewCreated()` Function

A fragment also typically has an `onViewCreated()` function. This will be called after `onCreateView()`, and it is where you configure and populate the widgets that are part of the UI that the fragment is managing.

In our case, we are using data binding, so this is where we can bind our `ToDoModel` into the layout, to invoke the binding expressions and fill in the widgets.

To do that, we:

- Get our `DisplayViewModel` from a `ViewModelProvider` tied to our fragment
- Get the model ID out of the arguments Bundle
- Pass that ID to `getModel()` on the `DisplayViewModel` to retrieve our `ToDoModel`
- Put that model, and a formatted edition of the `createdOn` value, into the binding

```
@Override
public void onCreateView(@NonNull View view,
                        @Nullable Bundle savedInstanceState) {
    super.onCreateView(view, savedInstanceState);

    DisplayViewModel vm =
        new ViewModelProvider(this).get(DisplayViewModel.class);
    String modelId = getArguments().getString(ARG_MODEL_ID);

    if (modelId == null) {
        throw new IllegalArgumentException("no modelId provided!");
    }

    ToDoModel model = vm.getModel(modelId);

    if (model != null) {
        binding.setModel(model);
        binding.setCreatedOnFormatted(DateUtils.getRelativeDateTimeString(
            getActivity(),
            model.createdOn.toEpochMilli(), DateUtils.MINUTE_IN_MILLIS,
            DateUtils.WEEK_IN_MILLIS, 0
        ));
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

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). Specifically, we are using the `getRelativeDateTimeString()` method on `DateUtils`, which will return a value that expresses the creation time relative to now using phrases like “5 minutes ago” or “2 days ago”.

The net result is that after `onViewCreated()` returns, our widgets will have the desired contents.

Of course, so far, we have glossed over the issue of when and how any of this happens. We have this lovely factory function... but something needs to call it. That function creates a perfectly delightful fragment... but something needs to arrange to show it on the screen. We will explore those steps in the upcoming sections.

The ListFragment

We also have a `ListFragment` that is responsible for displaying the list of to-do items. When the user clicks on an item in the list, we want to then show the `DisplayFragment`.

There are several pieces that come together to show this list, with `ListFragment` acting as a central coordinator for all of them.

The Fragment Layout

As we have seen in other samples in this book, this sample app uses a `RecyclerView` for displaying the list of to-do items. Our `todo_roster` layout resource has that `RecyclerView`, and nothing else:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView android:id="@+id/items"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

(from [FragmentManual/src/main/res/layout/todo_roster.xml](#))

The Row Layout

We also need a layout resource for the rows to appear in the list. That layout — `todo_row` — is a bit more interesting:

```
<?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="model"
            type="com.commonware.jetpack.sampler.fragments.ToDoModel" />

    </data>
```



```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?android:attr/selectableItemBackground"
    android:clickable="true"
    android:focusable="true">

    <CheckBox
        android:id="@+id/checkbox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:checked="@{model.isCompleted}"
        android:textSize="@dimen/desc_size"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="@{model.description}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/checkbox"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [FragmentManual/src/main/res/layout/todo_row.xml](#))

From a user experience standpoint, we want:

- The user to be able to mark the to-do item as completed by toggling the checkbox (though, in this version of the app, we will not try to save that change)
- The user to be able to view details of the item, via `DisplayFragment`, by clicking elsewhere on the row

Normally, a `CheckBox` is both the actual “checkbox” square plus associated text. So, visually, we could say that each row is just a `CheckBox` widget. But then clicks on the text will wind up toggling the `CheckBox` state, as `CheckBox` interprets all clicks on it equally. In our case, we want to distinguish between clicks on the checkbox and clicks on the rest of the row, so we cannot just use `CheckBox` alone.

To that end, we have both a `CheckBox` and a `TextView` in our row, wrapped in a `ConstraintLayout`, to mimic the presentation of a regular `CheckBox` but not have clicks on the text trigger `CheckBox` state changes.

The root `ConstraintLayout` has its attributes set up such that it represents something that can be clicked upon:

- `android:background="?android:attr/selectableItemBackground"`
- `android:clickable="true"`
- `android:focusable="true"`

We are also using data binding here, to populate the `CheckBox` checked state and the `TextView` text based on the data in a particular `ToDoModel`.

The Viewmodel

Once again, we have a `ViewModel` to hold our data to be displayed. In this case, `ListViewModel` holds the list of items obtained from the `ToDoRepository`:

```
package com.commonware.jetpack.samplerj.fragments;

import java.util.List;
import androidx.lifecycle.ViewModel;

public class ListViewModel extends ViewModel {
    final List<ToDoModel> items = ToDoRepository.INSTANCE.getItems();
}
```

(from [FragmentManual/src/main/java/com/commonware/jetpack/samplerj/fragments/ListViewModel.java](#))

```
package com.commonware.jetpack.sampler.fragments

import androidx.lifecycle.ViewModel

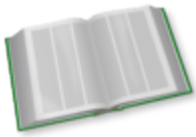
class ListViewModel : ViewModel() {
    val items = ToDoRepository.getItems()
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ListViewModel.kt](#))

The Adapter and Row Holder

Since we are using a RecyclerView, we need a RecyclerView.Adapter and a RecyclerView.ViewHolder. And we have an interesting requirement: we want our ListFragment to get control when the user clicks on those rows, so we can navigate to the DisplayFragment.

In Kotlin, we can accomplish this by having our adapter and view-holder accept a function type as a parameter, so we can call that function when the user clicks on a row.



You can learn more about Kotlin immutability, functional programming, and function types in the "Functional Programming" chapter of [Elements of Kotlin](#)!

In Java, we can create a custom listener interface and use instances of that listener in lieu of the Kotlin function type.

Our RecyclerView.ViewHolder is called `ToDoListRowHolder`:

```
package com.commonsware.jetpack.samplerj.fragments;

import com.commonsware.jetpack.samplerj.fragments.databinding.TODORowBinding;
import androidx.recyclerview.widget.RecyclerView;

class ToDoListRowHolder extends RecyclerView.ViewHolder {
    interface OnRowClickListener {
        void onRowClick(ToDoModel model);
    }

    private final TODORowBinding binding;

    ToDoListRowHolder(TODORowBinding binding, OnRowClickListener listener) {
        super(binding.getRoot());

        this.binding = binding;

        binding.getRoot().setOnClickListener(v -> {
            if (binding.getModel() != null) {
                listener.onRowClick(binding.getModel());
            }
        })
    }
}
```

ADOPTING FRAGMENTS

```
});  
}  
  
void bind(TodoModel model) {  
    binding.setModel(model);  
    binding.executePendingBindings();  
}  
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoListRowHolder.java](#))

```
package com.commonsware.jetpack.sampler.fragments  
  
import androidx.recyclerview.widget.RecyclerView  
import com.commonsware.jetpack.sampler.fragments.databinding.TodoRowBinding  
  
class ToDoListRowHolder(  
    private val binding: TodoRowBinding,  
    onClick: (TodoModel) -> Unit  
) :  
    RecyclerView.ViewHolder(binding.root) {  
    init {  
        binding.root.setOnClickListener { binding.model?.let { onClick(it) } }  
    }  
  
    fun bind(model: TodoModel) {  
        binding.model = model  
        binding.executePendingBindings()  
    }  
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoListRowHolder.kt](#))

We receive a `TodoRowBinding` object in our constructor — that is the data binding generated binding class for our `todo_row` layout resource. We also receive the Kotlin function type or the Java `OnClickListener` to use to report when the user clicks on rows.

Then, we:

- Hold onto the binding in a property
- Set up a click listener on the `ConstraintLayout` (the root of our binding) and invoke the Kotlin function type or call the Java listener when the user clicks on a row
- Bind a model to the binding in `bind()` as requested

ADOPTING FRAGMENTS

We can supply the `ToDoModel` to the function type or listener by asking our binding for its model via `getModel()`. This way, we can pass the `ToDoModel` upstream when the user clicks on its row.

Our `RecyclerView.Adapter` is called `ToDoListAdapter`:

```
package com.commonware.jetpack.samplerj.fragments;

import android.view.LayoutInflater;
import android.view.ViewGroup;
import com.commonware.jetpack.samplerj.fragments.databinding.TODORowBinding;
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.ListAdapter;

class ToDoListAdapter extends ListAdapter<ToDoModel, ToDoListRowHolder> {
    private final LayoutInflater inflater;
    private final ToDoListRowHolder.OnRowClickListener listener;

    protected ToDoListAdapter(LayoutInflater inflater,
                               ToDoListRowHolder.OnRowClickListener listener) {
        super(TODOModel.DIFF_CALLBACK);
        this.inflater = inflater;
        this.listener = listener;
    }

    @NonNull
    @Override
    public ToDoListRowHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                                int viewType) {
        return new ToDoListRowHolder(
            TODORowBinding.inflate(inflater, parent, false), listener);
    }

    @Override
    public void onBindViewHolder(@NonNull ToDoListRowHolder holder,
                                int position) {
        holder.bind(getItem(position));
    }
}
```

(from [FragmentManual/src/main/java/com/commonware/jetpack/samplerj/fragments/ToDoListAdapter.java](#))

```
package com.commonware.jetpack.sampler.fragments;

import android.view.LayoutInflater;
import android.view.ViewGroup;
import androidx.recyclerview.widget.ListAdapter;
import com.commonware.jetpack.sampler.fragments.databinding.TODORowBinding
```

ADOPTING FRAGMENTS

```
class ToDoListAdapter(private val inflater: LayoutInflater,
    private val onClick: (ToDoModel) -> Unit) :
    ListAdapter<ToDoModel, ToDoListRowHolder>(ToDoModelDiffCallback) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        ToDoListRowHolder(ToDoRowBinding.inflate(inflater, parent, false), onClick)

    override fun onBindViewHolder(holder: ToDoListRowHolder, position: Int) {
        holder.bind(getItem(position))
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoListAdapter.kt](#))

ToDoListAdapter is a ListAdapter, and so we need our DiffUtil.ItemCallback to help identify changes in our data model. That is implemented on ToDoModel, as our model class should know how to compare instances of itself:

```
static final DiffUtil.ItemCallback<ToDoModel> DIFF_CALLBACK =
    new DiffUtil.ItemCallback<ToDoModel>() {
        @Override
        public boolean areItemsTheSame(@NonNull ToDoModel oldItem,
                                         @NonNull ToDoModel newItem) {
            return oldItem == newItem;
        }

        @Override
        public boolean areContentsTheSame(@NonNull ToDoModel oldItem,
                                           @NonNull ToDoModel newItem) {
            return oldItem.isCompleted == newItem.isCompleted &&
                oldItem.description.equals(newItem.description);
        }
    };
};
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoModel.java](#))

```
object ToDoModelDiffCallback : DiffUtil.ItemCallback<ToDoModel>() {
    override fun areItemsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem === newItem

    override fun areContentsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.isCompleted == newItem.isCompleted &&
            oldItem.description == newItem.description
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoModel.kt](#))

ToDoListAdapter also accepts our function type or listener in its constructor, to pass to the constructors of each of the ToDoListRowHolder instances.

Beyond that, `ToDoListAdapter` is a fairly simple `ListAdapter` implementation, creating instances of `ToDoListViewHolder` in `onCreateViewHolder()` and binding `ToDoModel` instances in `onBindViewHolder()`.

The Fragment

With all that done, our `ListFragment` has little to do, other than connect the remaining pieces:

```
package com.commonware.jetpack.samplerj.fragments;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.commonware.jetpack.samplerj.fragments.databinding.TODORosterBinding;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.fragment.app.Fragment;
import androidx.lifecycle.ViewModelProvider;
import androidx.recyclerview.widget.LinearLayoutManager;

public class ListFragment extends Fragment {
    private TODORosterBinding binding;

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                             @Nullable ViewGroup container,
                             @Nullable Bundle savedInstanceState) {
        binding = TODORosterBinding.inflate(inflater, container, false);

        return binding.getRoot();
    }

    @Override
    public void onViewCreated(@NonNull View view,
                              @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        ListViewModel vm = new ViewModelProvider(this).get(ListViewModel.class);

        binding.items.setLayoutManager(new LinearLayoutManager(getContext()));

        ToDoListAdapter adapter = new ToDoListAdapter(getLayoutInflater(),
            this::navTo);

        adapter.submitList(vm.items);
        binding.items.setAdapter(adapter);
    }

    @Override
    public void onDestroyView() {
        super.onDestroyView();
    }
}
```

ADOPTING FRAGMENTS

```
binding = null;
}

private void navTo(TodoModel model) {
    getParentFragmentManager().beginTransaction()
        .replace(android.R.id.content, DisplayFragment.newInstance(model.id))
        .addToBackStack(null)
        .commit();
}
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](https://github.com/commonsware/jetpack-samplerj/blob/master/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java))

As with `DisplayFragment`, we will focus on the Java edition of the fragment now and will examine the Kotlin edition later in the chapter.

Our `onCreateView()` function sets up the UI to be managed by this fragment. That is the `RecyclerView` defined in the `todo_roster` layout resource. That layout resource does not use data binding, so we can use `inflate()` on `LayoutInflater` directly to load that layout resource and create the `RecyclerView`.

As with `DisplayFragment`, our `onViewCreated()` function configures the UI that we inflated in `onCreateView()`. Here we:

- Get our `ListViewModel` from a `ViewModelProvider`
- Create a `LinearLayoutManager` and attach it to the `RecyclerView`
- Create a `ToDoListAdapter` and attach it to the `RecyclerView`
- Get our list of `ToDoModel` objects from the `ListViewModel` and supply those to the `ToDoListAdapter` via `submitList()`

When we create the `ToDoListAdapter`, we need to supply our `onItemClickListener` function type or listener implementation. In Java, we use a Java 8 method reference to have a generated listener call our `navTo()` method.

So, when the user clicks a row, whether we are in Java or Kotlin, `navTo()` is called. There, we need to arrange to show the `DisplayFragment`.

The FragmentTransaction

`navTo()` uses a `FragmentTransaction` to display the `DisplayFragment`.

The Java edition of `navTo()` has a fairly classic recipe for performing a `FragmentTransaction`:


```
private void navTo(TodoModel model) {
    getParentFragmentManager().beginTransaction()
        .replace(android.R.id.content, DisplayFragment.newInstance(model.id))
        .addToBackStack(null)
        .commit();
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](#))

First, to get a `FragmentManager`, we need a `FragmentManager`. A fragment can get a `FragmentManager` by calling `getParentFragmentManager()`. This returns the `FragmentManager` that manages calling `Fragment` and any of its peer fragments.

Next, to get a `FragmentManager` for use, we can call `beginTransaction()` on the `FragmentManager`. This returns a `FragmentManager` ready for us to configure and commit.

In our case, we do two things to configure the `FragmentManager`:

1. We call `replace()` to say “please replace any fragment in the `android.R.id.content` container with this new fragment”. The new fragment is our `DisplayFragment`, created using the factory method. The container is `android.R.id.content`, which is the framework’s ID for the container for an activity’s UI. When we call `setContentView()` on an activity, the “content view” winds up as a child of the `android.R.id.content` container.
2. We call `addToBackStack(null)`, indicating that when the user clicks the BACK button, we want the `FragmentManager` to be rolled back. In our case, that would revert the `replace()` call, which would cause `android.R.id.content` to hold whatever fragment preceded the `DisplayFragment`. As we will see shortly, the `ListFragment` itself is going in that container, so reverting that transaction will cause the `ListFragment` to be displayed again.

Finally, once our `FragmentManager` is configured, we call `commit()` to say “make it so”. Shortly, the `DisplayFragment` will be shown on the screen, replacing the `ListFragment`.

The Activity

All that remains is for our activity to set up the `ListFragment` to be shown. That too is handled via a `FragmentManager`, much like how we navigated from

ADOPTING FRAGMENTS

ListFragment to DetailFragment. However, there is one wrinkle: we only execute the `FragmentManager` if our `savedInstanceState` is null:

```
package com.commonware.jetpack.samplerj.fragments;

import android.os.Bundle;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (savedInstanceState == null) {
            getSupportFragmentManager().beginTransaction()
                .add(android.R.id.content, new ListFragment())
                .commit();
        }
    }
}
```

(from [FragmentManual/src/main/java/com/commonware/jetpack/samplerj/fragments/MainActivity.java](#))

```
package com.commonware.jetpack.sampler.fragments

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.commit

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        if (savedInstanceState == null) {
            supportFragmentManager.commit {
                add(android.R.id.content, ListFragment())
            }
        }
    }
}
```

(from [FragmentManual/src/main/java/com/commonware/jetpack/sampler/fragments/MainActivity.kt](#))

When the device undergoes a configuration change, Android will automatically destroy and recreate the activity *and its fragments*. As a result, in the re-created

activity, by the time `onCreate()` is called, any displayed fragments will already exist. In that case, we do not need to create them ourselves. So, to distinguish between the first time the activity is created and when the activity is re-created after a configuration change, we see if the passed-in state `Bundle` is `null`. If it is, then our fragment should not yet exist, so we create one and `add()` it to the container via a `FragmentTransaction`. If the `Bundle` is not `null`, though, then the fragment should already exist, so we can skip the step for creating and adding it.

Another approach that we could use is to see if the container already has a fragment in it, using code like:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    if (supportFragmentManager.findFragmentById(android.R.id.content) == null) {
        supportFragmentManager.transaction {
            add(android.R.id.content, ListFragment())
        }
    }
}
```

Those approaches are subtly different. The original asks “did the activity get re-created?”. The revision asks “do we have a fragment in the container already?”. In the context of this app, those will be equivalent, so either approach can work. In more elaborate apps, where fragments might be coming and going with greater frequency and complexity, those approaches may yield different results, and you will want to choose the one that better matches the actual need of your code and situation.

On an activity, we use `getSupportFragmentManager()` to get the `FragmentManager` to use. There is also a `getFragmentManager()` method, but it refers to the deprecated framework edition of fragments, which is not what we want. `getFragmentManager()` itself is deprecated, to help further warn us that this is not the method that we are looking for.

The Kotlin code is a bit different, because the Kotlin edition of the project has added `androidx.fragment:fragment-ktx` as a dependency. This Android KTX library offers some simpler syntax for Kotlin’s use of `FragmentTransaction`.

Part of that syntax change is just ordinary Kotlin, where `getSupportFragmentManager()` can be replaced by reading the `supportFragmentManager` pseudo-property.

Part of that syntax change comes from Android KTX, in the form of that `androidx.fragment:fragment-ktx` library. It adds an extension function to `FragmentManager`, named `commit()`. The `commit()` function:

- Accepts a function type as a parameter, typically in the form of a lambda expression as shown here
- Begins a `FragmentTransaction`
- Invokes that function type, setting the `FragmentTransaction` to be the current object (i.e., `this`), so calls like `replace()` and `addToBackStack()` can be called directly
- Commits that `FragmentTransaction` once the function type returns

The Recap

So, we have two fragments, each with their own viewmodel. The `MainActivity` sets up the first fragment (`ListFragment`). That fragment renders its own UI and switches to a different fragment (`DisplayFragment`) based on user input. The second fragment handles *its* own UI, independently from the first fragment, other than for data explicitly passed between them (the to-do item's ID).

We could have implemented the same thing using two activities, instead of a single activity and two fragments. A lot of classic Android app development would have taken that approach. The Jetpack recommendations are to use fragments where possible and minimize your number of activities, for greater flexibility in implementing your UI.

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 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. For example, you need to attach an adapter to a `ListFragment` — a common place to do that is in `onViewCreated()`, as you know that the `ListView` is set up at that time.

`onActivityCreated()` will be called 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.

ADOPTING FRAGMENTS

Google offers this diagram, depicting the order in which these events will occur:

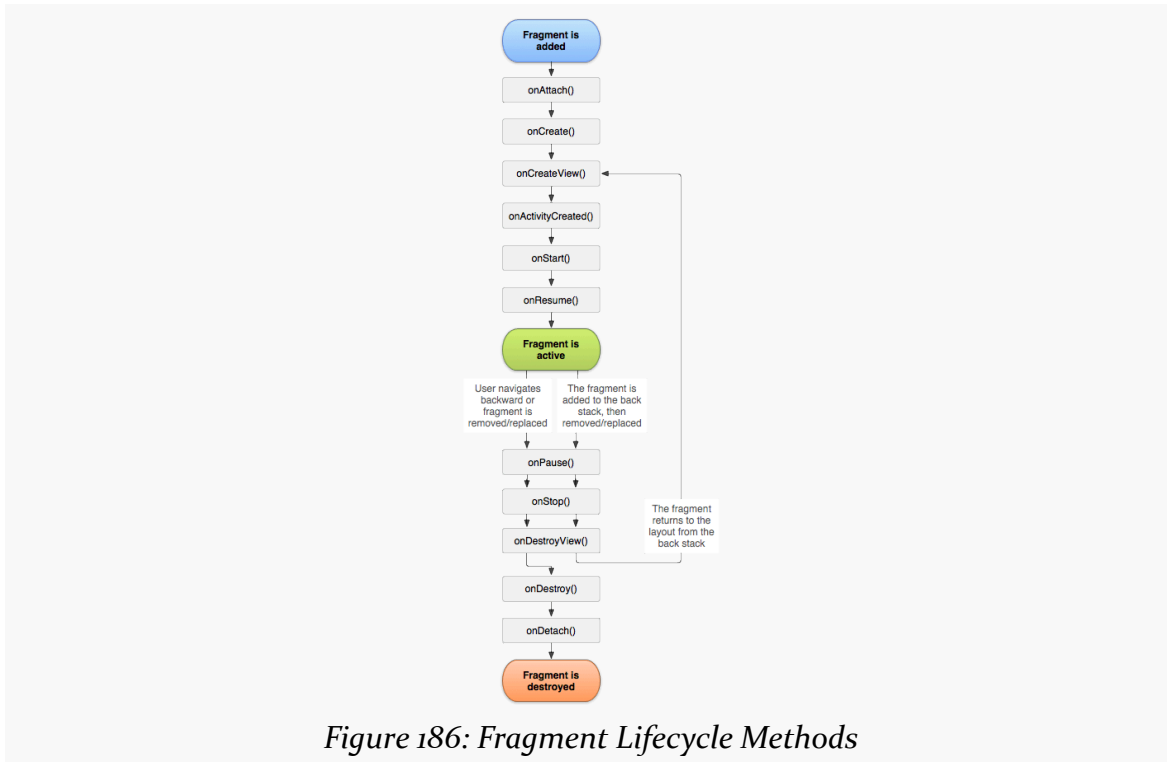


Figure 186: Fragment Lifecycle Methods

(the above image is 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](#))

View Binding and Fragments

A fragment may outlive its views. It is possible for a fragment to:

- Be created and have `onCreate()` be called
- Have `onCreateView()` and `onViewCreated()` be called
- Have `onDestroyView()` be called
- Have `onCreateView()` and `onViewCreated()` be called *again* (e.g., the fragment gets re-displayed)
- Have `onDestroyView()` be called
- Have `onDestroy()` be called

In between the first `onDestroyView()` and the second `onCreateView()`, ideally our

fragment has no references to the views that were created and destroyed from the first `onCreateView()/onDestroyView()` cycle. In the specific case of view binding, this means that we do not want to hold onto our binding object after `onDestroyView()` gets called.

The recommended pattern is to set the Java field or Kotlin property that holds the binding object to null in `onDestroyView()`. The mechanics of this will vary a bit between Java and Kotlin, owing to Kotlin caring deeply about null.

Java

The Java edition of `ListFragment` has a binding field:

```
private TodoRosterBinding binding;
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](#))

We then set a value to it in `onCreateView()`, using `TodoRosterBinding.inflate()`:

```
@Nullable
@Override
public View onCreateView(@NonNull LayoutInflater inflater,
                        @Nullable ViewGroup container,
                        @Nullable Bundle savedInstanceState) {
    binding = TodoRosterBinding.inflate(inflater, container, false);

    return binding.getRoot();
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](#))

We can then use it in places like `onViewCreated()`:

```
@Override
public void onViewCreated(@NonNull View view,
                        @Nullable Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    ListViewModel vm = new ViewModelProvider(this).get(ListViewModel.class);

    binding.items.setLayoutManager(new LinearLayoutManager(getContext()));

    ToDoListAdapter adapter = new ToDoListAdapter(getLayoutInflater(),
        this::navTo);
}
```

ADOPTING FRAGMENTS

```
adapter.submitList(vm.items);
binding.items.setAdapter(adapter);
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](#))

Finally, we can set the field back to null in `onDestroyView()`:

```
@Override
public void onDestroyView() {
    super.onDestroyView();

    binding = null;
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](#))

As a result, after `onDestroyView()`, the widgets created in `onCreateView()` can be garbage-collected.

Kotlin

The Java approach results in a fair bit of boilerplate. The Kotlin example isolates a lot of that in a common `ViewBindingFragment` class.

ViewBindingFragment

`ViewBindingFragment` resides in the Utilities module that we examined [in a previous chapter](#). It handles inflating a layout using a view binding for you, as well as setting that view binding's property to null in `onDestroyView()`:

```
package com.commonsware.jetpack.sampler.util

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.viewbinding.ViewBinding

abstract class ViewBindingFragment<Binding : ViewBinding>(
    private val bindingInflater: (LayoutInflater, ViewGroup?, Boolean) -> Binding
) : Fragment() {
    private var binding: Binding? = null

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) {
```


ADOPTING FRAGMENTS

```
) : View {
    return bindingInflater(inflater, container, false)
        .apply { binding = this }
        .root
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

protected fun requireBinding(): Binding = binding
    ?: throw IllegalStateException("You used the binding before onCreateView() or after onDestroyView()")

protected fun useBinding(bindingUse: (Binding) -> Unit) {
    bindingUse(requireBinding())
}
}
```

(from [Utilities/src/main/java/com/commonsware/jetpack/sampler/util/ViewBindingFragment.kt](#))

ViewBindingFragment uses generics. The Binding generic type is the generated view binding class associated with the layout for this fragment (e.g., `TodoRosterBinding`). Those classes all extend from a Jetpack-supplied `ViewBinding` base class, so Binding is declared as being a sub-type of `ViewBinding`.



You can learn more about Kotlin generics in the "Generics" chapter of [Elements of Kotlin](#)!

The constructor takes a function type as a parameter. The odd-looking function signature is the one used by the `inflate()` functions on the generated `ViewBinding` classes. So, subclasses need to pass a function reference to that `inflate()` function:

```
class ListFragment :
    ViewBindingFragment<TodoRosterBinding>(TodoRosterBinding::inflate) {
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ListFragment.kt](#))

`ViewBindingFragment` then has its own `onCreateView()` and `onDestroyView()` functions. `onCreateView()` calls that `bindingInflater` function type to inflate the view binding, holds the value in the `binding` property, and returns the root `View` to satisfy the `onCreateView()` contract. `onDestroyView()` simply sets `binding` to be `null`.

However, `binding` is private. Subclasses can use it in one of two ways:

- Call `requireBinding()` to get the view binding object... or throw an exception if you called it sometime outside of the `onCreateView()/onDestroyView()` pair
- Call `useBinding()`, passing in a lambda expression or other function type, which turns around and invokes your lambda expression with the view binding object

ListFragment

As a result, subclasses of `ViewBindingFragment` can focus on their business logic and let `ViewBindingFragment` take care of inflating and clearing the view binding object.

`ListFragment`, for example, has `ViewBindingFragment` manage the `TodoRosterBinding` and just works with its widgets:

```
package com.commonware.jetpack.sampler.fragments

import android.os.Bundle
import android.view.View
import androidx.fragment.app.commit
import androidx.fragment.app.viewModels
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonware.jetpack.sampler.fragments.databinding.TodoRosterBinding
import com.commonware.jetpack.sampler.util.ViewBindingFragment

class ListFragment :
    ViewBindingFragment<TodoRosterBinding>(TodoRosterBinding::inflate) {
    private val vm: ListViewModel by viewModels()

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        useBinding { binding ->
            binding.items.layoutManager = LinearLayoutManager(context)

            val adapter = ToDoListAdapter(layoutInflater) {
                navTo(it)
            }

            binding.items.adapter = adapter.apply { submitList(vm.items) }
        }

        private fun navTo(model: ToDoModel) {
            parentFragmentManager.commit {
```

ADOPTING FRAGMENTS

```
        replace(android.R.id.content, DisplayFragment.newInstance(model.id))
        addToBackStack(null)
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ListFragment.kt](#))

DisplayFragment

Similarly, DisplayFragment lets ViewBindingFragment manage the TodoDisplayBinding object:

```
package com.commonsware.jetpack.sampler.fragments

import android.os.Bundle
import android.text.format.DateUtils
import android.view.View
import androidx.core.os.bundleOf
import androidx.fragment.app.viewModels
import com.commonsware.jetpack.sampler.fragments.databinding.TodoDisplayBinding
import com.commonsware.jetpack.sampler.util.ViewBindingFragment

private const val ARG_MODEL_ID = "modelId"

class DisplayFragment :
    ViewBindingFragment<TodoDisplayBinding>(TodoDisplayBinding::inflate) {
    companion object {
        fun newInstance(modelId: String) = DisplayFragment().apply {
            arguments = bundleOf(ARG_MODEL_ID to modelId)
        }
    }

    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        super.onCreateView(view, savedInstanceState)

        val vm: DisplayViewModel by viewModels()
        val model = vm.getModel(
            arguments?.getString(ARG_MODEL_ID)
            ?: throw IllegalStateException("no modelId provided!")
        )

        model?.let {
            useBinding { binding ->
                binding.model = model

                binding.createdOnFormatted = DateUtils.getRelativeDateTimeString(
```

```
        activity,
        model.createdOn.toEpochMilli(), DateUtils.MINUTE_IN_MILLIS,
        DateUtils.WEEK_IN_MILLIS, 0
    )
}
}
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/DisplayFragment.kt](https://github.com/commonsware/jetpack/sampler/fragments/DisplayFragment.kt))

Context **Anti-Pattern: Assuming Certain Types**

Fragment has a `getContext()` method that you can call, to retrieve a Context, should you need one for retrieving resources, etc.

It is likely that this Context is the Activity that hosts the fragment. However you should not *assume* that this Context is the Activity that hosts the fragment. Google has been preparing for cases where fragments might be owned by something else, though what that “something else” is has yet to be determined.

If the documentation for the source of a Context stipulates that it is a particular type of Context, it should be safe to treat it as such, downcasting it as needed (e.g., context as Activity in Kotlin). Otherwise, limit your use of that Context to things that are available on Context and make no assumptions about its concrete type. That concrete type might vary from version to version of Android, or even between devices on the same Android version (due to manufacturer tweaks to Android).

Navigating Your App

In the beginning, if we wanted multiple screens of content, we would use multiple activities. To switch from screen to screen, we would use `startActivity()`.

As Android evolved, many developers started to use fragments. To switch from screen to screen, we would execute a `FragmentTransaction`.

All of that works and you are welcome to use it.

However, as part of Jetpack, Google is pushing a new Navigation component. This serves as a replacement for many direct uses of `FragmentTransaction` and `startActivity()`, while still allowing you to navigate from activities and fragments to other activities and fragments.

What We Get from the Navigation Component

The Navigation component does not add any new capabilities to Android. It can't, after all — it comes from a library, and anything that can be done by Library A could be done by Library B or Hand-Written-Code C. Anything you can do with the Navigation component you can do “the old-fashioned way” using `startActivity()`, `FragmentTransaction`, and related techniques.

So... why bother with the Navigation component?

Uniform API/Isolation of Details

When we call `startActivity()`, we have to provide an `Intent` that identifies the activity that we want to start. When we commit a `FragmentTransaction`, we have to configure that object with an instance of the fragment to be displayed. As a result:

- Our code is tied to a specific type of navigation (activity or fragment)
- Our code knows implementation details of the destination (the Java/Kotlin class that is the target)

A big thing in computer programming is “separation of concerns”. We want one chunk of code to know as little as possible about the next chunk of code, in case we need to radically change the implementation of either chunk. With classic manual navigation, our activities and fragments know more about the other activities and fragments than is ideal.

With the Navigation component, we start to get away from that.

One key aspect is that all navigation is the same with the Navigation component. When we want to switch to a different screen, we use the same API regardless of whether that screen is implemented by an activity or a fragment. Hence, we want to switch implementations someday (e.g., convert an activity to a fragment), the code that navigates to that changed destination may not itself need to change.

Another aspect is that the details of how different screens are implemented is contained in a new type of resource (navigation). The resource contains details about classes and whether they represent activities or fragments. The resource uses IDs — akin to widget IDs — to identify screens and how we get there. Our Java/Kotlin code uses those IDs to tell the Navigation component where we want to go. So, now not only do we use one API for navigating to both activities and fragments, but the details of *which* activities and fragments are hidden behind an ID. If we want to swap out one implementation of a screen with another, those navigating to that screen should not need to change.

Graphical Representation of Flows

The navigation resource type, like most of Android’s resources, is an XML file with a particular set of elements. And, like layout and menu resources, Android Studio offers a graphical editor for navigation resources, allowing you to connect screens using a drag-and-drop interface:

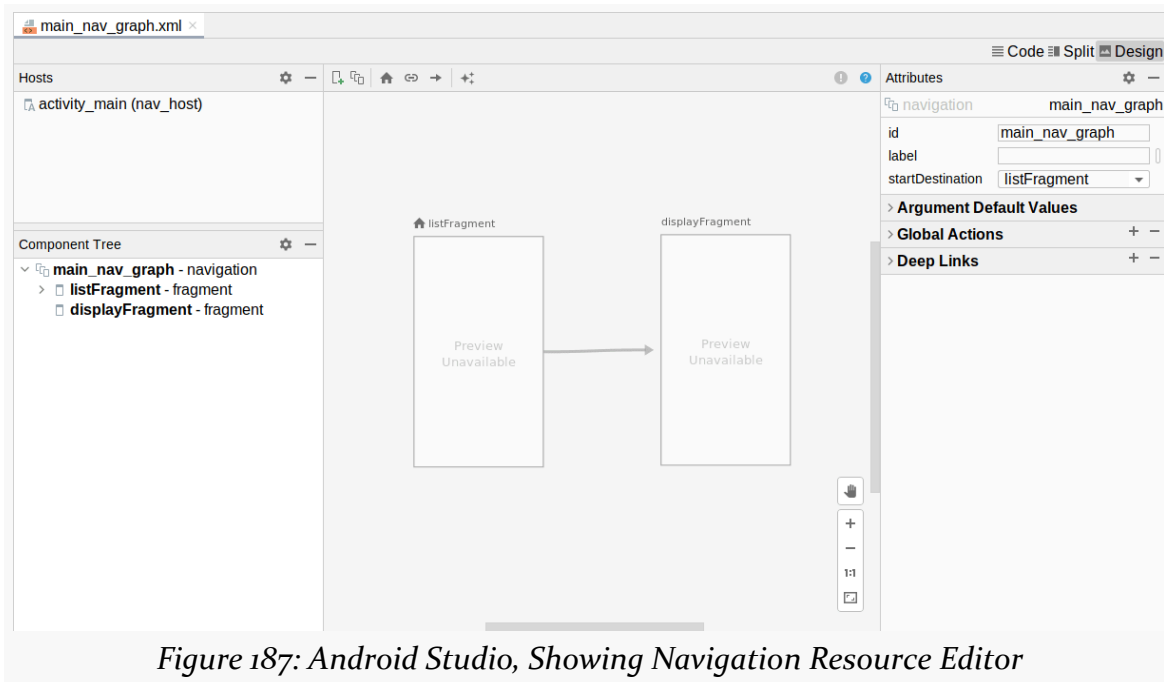


Figure 187: Android Studio, Showing Navigation Resource Editor

We will explore the use of this editor later in this chapter.

“Safe Args”

In both [the chapter on activities](#) and [the chapter on fragments](#), we saw how we can pass data from one screen to another. In the case of activities, that is in the form of Intent extras. In the case of fragments, that is in the form of the arguments Bundle. As it turns out, the extras in an Intent are contained in a Bundle, so in both forms of manual navigation, your data is passed via a Bundle.

The problem with a Bundle is that while a Bundle supports a range of data types for values, any key could have any of those data types. This can lead to problems, if the sender specifies a String where the recipient is expecting an Int, or something like that. It is as if we got rid of the strong typing available in Java/Kotlin and just held onto all of our data in Map objects.

An add-on to the Navigation component helps with this by offering “Safe Args”. Part of the details that we can put into the navigation resource is what data is expected by any given screen. Then, Android Studio will code-generate:

- A class to help code wishing to navigate to that screen provide the data, with the expected data types
- A class to help the implementation of the screen retrieve that data, once again with the expected data types

“Under the covers”, those generated classes work with Intent extras and the arguments Bundle, but those implementation details are hidden from us. This gives us type-safe navigation, so we do not accidentally provide data of the wrong type.

App Bar Up Integration

You may have noticed that in some apps, in some cases, a leftward-pointing arrow appears in the app bar:



Figure 188: Up Button in App Bar

This button allows the user to navigate “up” in the app. In most cases, this is the same as the user pressing the system BACK button in the navigation bar. Sometimes, in complicated apps with complicated navigation, up might lead somewhere else that represents “up” in some hierarchy of content.

The Navigation component allows you to tie your app bar (e.g., a Toolbar) into the navigation flow, such that the “up” button is shown when the Navigation component deems it to be appropriate.

Simpler Support of Advanced Features

Those reasons may be enough to warrant experimenting with the Navigation component.

For veteran Android developers, the Navigation component also helps to simplify the use of a variety of advanced features, such as:

- “Deep links”, to help Google’s search engine and other Web content to launch an app (via a link) and drive straight to some inner portion of that app, instead of just opening the launcher activity
- Transitions, a means by which we can give the appearance of a widget from one screen “moving” to another screen, such as a thumbnail image in a list “zooming” into a larger image when the user clicks on the list item and goes to a detail screen

Elements of Navigation

There are four main pieces of the Navigation component:

- The navigation resources
- The Android Studio editor for those resources
- The `NavHostFragment` for intra-activity navigation between fragments
- The `NavController` that lets you navigate at runtime

Navigation Resources

Google only infrequently creates a new type of resource, and almost never do they create one that is not handled directly by the OS.

For the Navigation component, they did just that.

You can have a `res/navigation/` directory in your module that holds navigation resources. Those are XML files with a root `<navigation>` element and a series of child elements that describe the navigation graph of your app.

Principally, the Navigation component handles navigation between fragments within an activity. Hence, the primary children of the `<navigation>` element are `<fragment>` elements, identifying a particular fragment that is part of the navigation graph. However, if you want to navigate to a different activity — perhaps one that has its own navigation graph — you can have child `<activity>` elements off of `<navigation>` that identify those.

The `<fragment>` and `<activity>` elements — otherwise known as the “destinations” — have a number of possible child elements, of which two are the most important:

- `<action>` indicates a destination that is supported from this screen; the Navigation component helps you navigate based on these actions
- `<argument>` indicates a particular bit of data that this screen is expecting as input

The root `<navigation>` element will have an `app:startDestination` attribute that identifies which `<fragment>` should be displayed when this navigation graph is used by an activity.

We will explore all of these in greater detail when we work through a sample app [a bit later in the chapter](#).

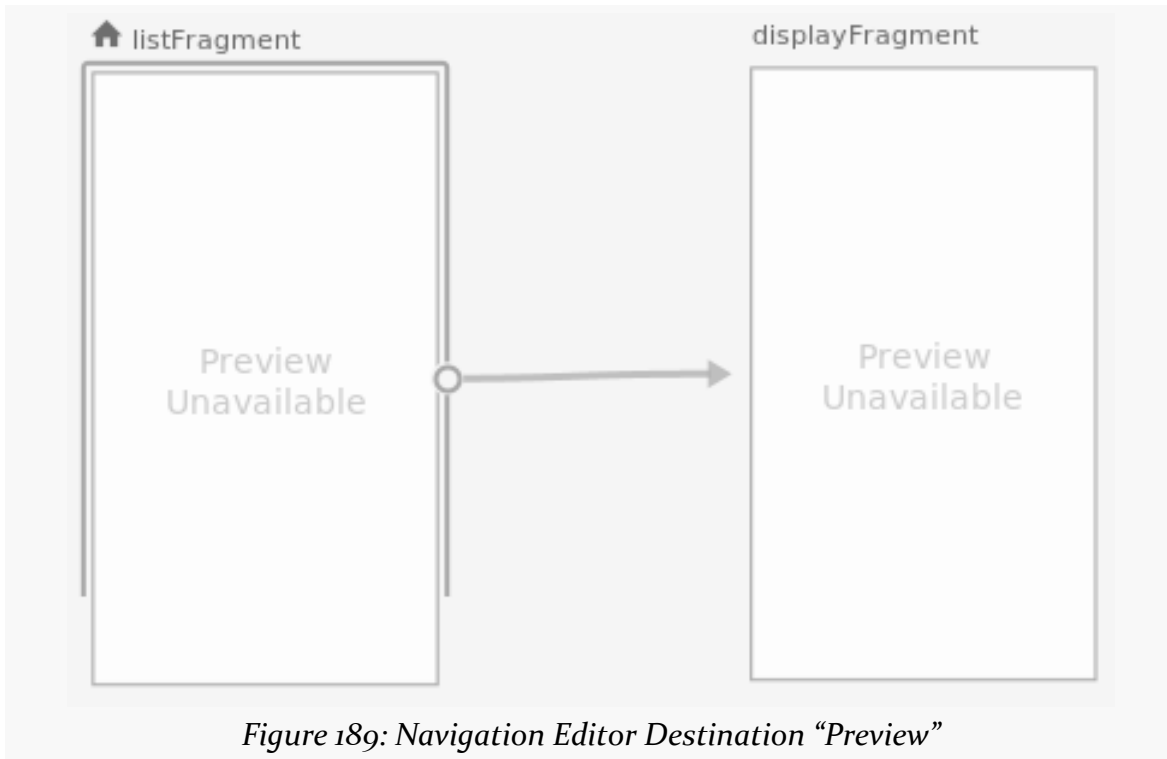
Navigation Resource Editor

You are welcome to maintain navigation resources manually by editing the XML. Or, you can use Android Studio's dedicated editor for those resources.

As with layout and menu resources, the navigation editor is a visual drag-and-drop editor, or you can work with the XML directly.

Destinations

The main “preview” area shows the existing destinations in the navigation graph:



To add a new destination, you can click the left-most toolbar button in the preview area:



NAVIGATING YOUR APP

This will bring up a list of all of the activities and fragments in your module, along with options for a “placeholder” and “create new destination”. The latter brings up a new-fragment wizard to help you create a new fragment.

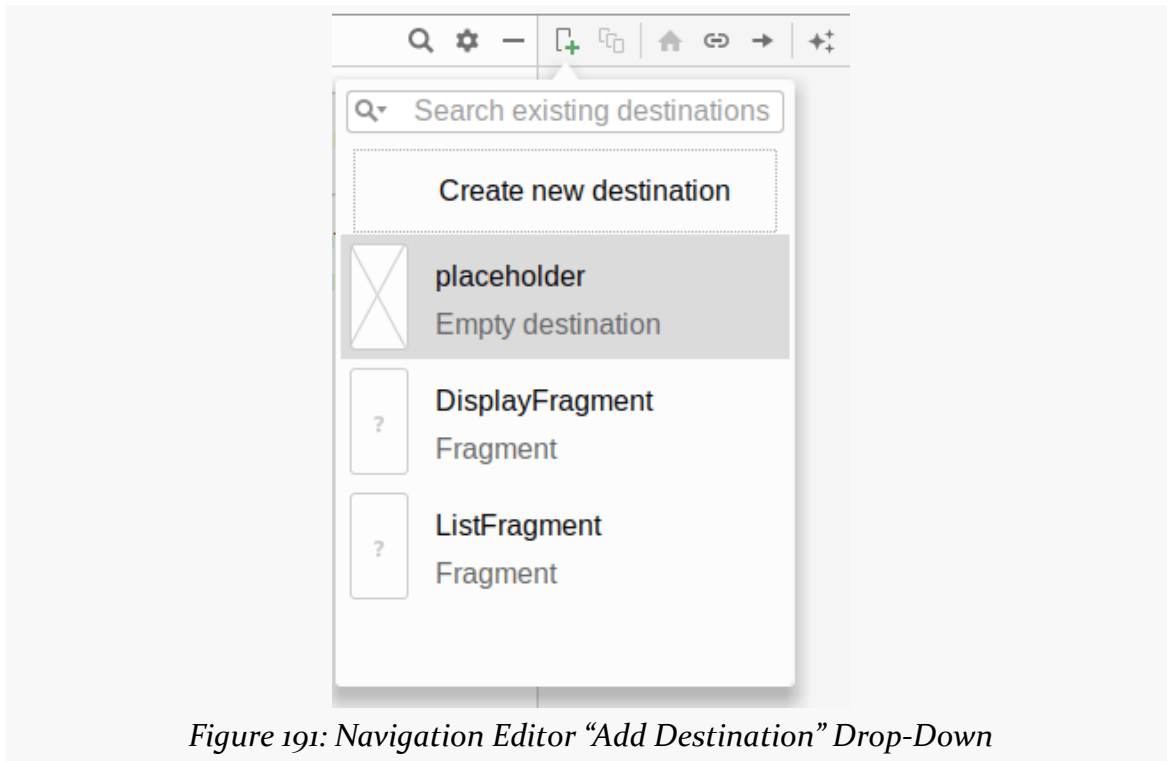
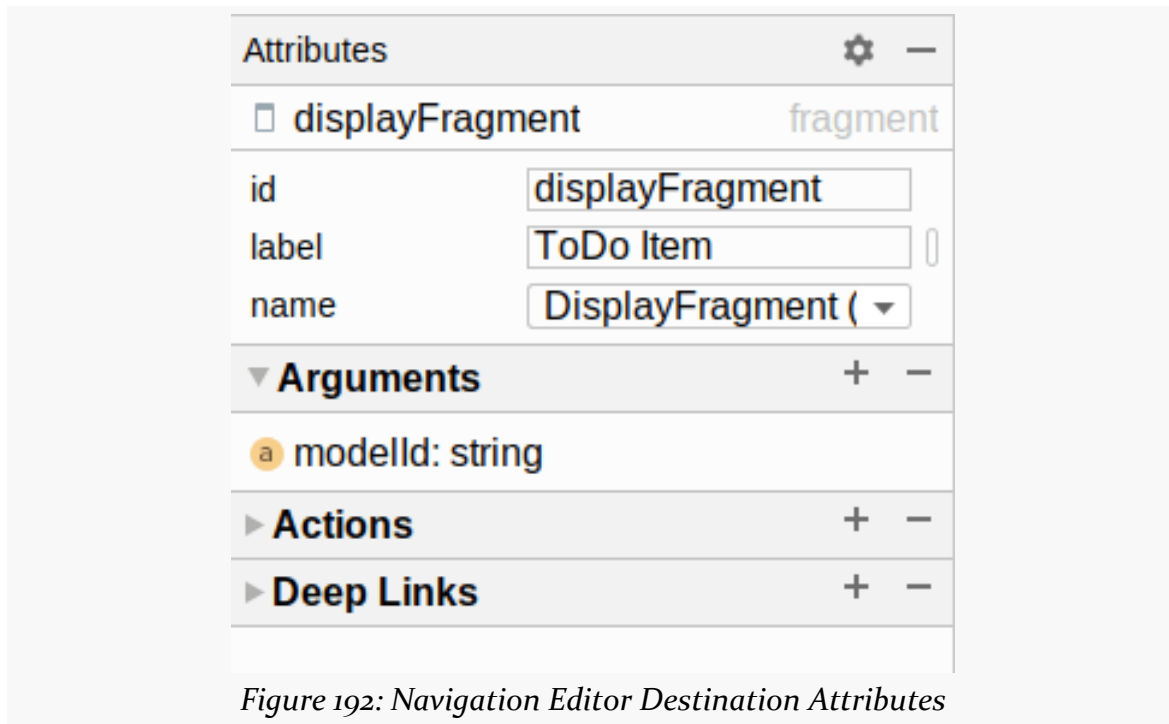


Figure 191: Navigation Editor “Add Destination” Drop-Down

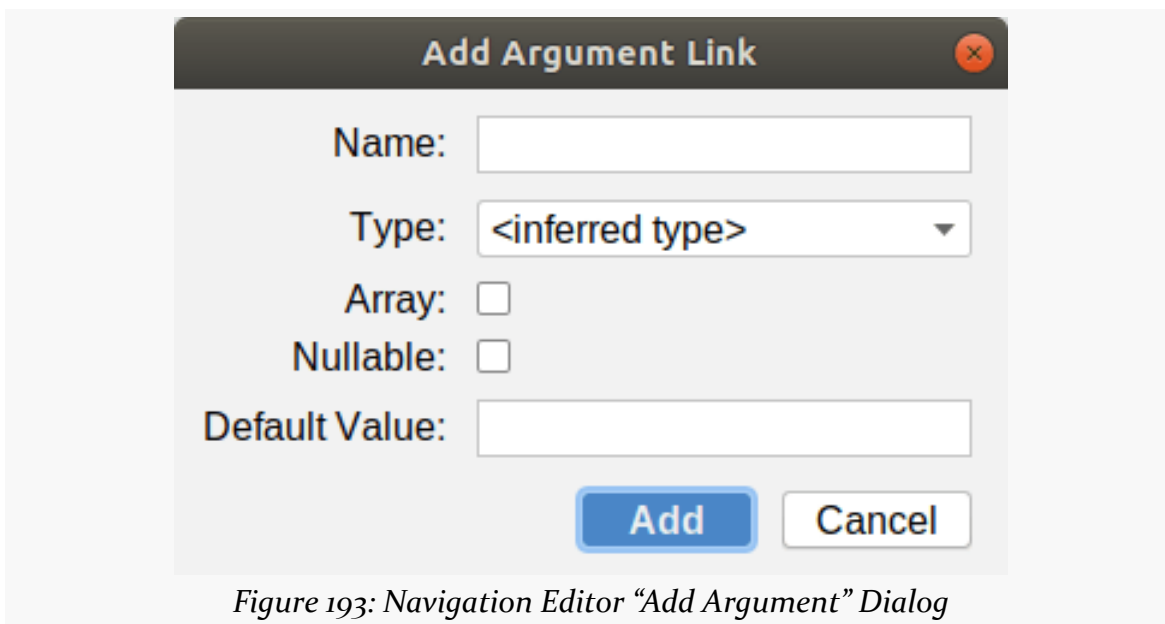
NAVIGATING YOUR APP

If you click on a particular destination, the “Attributes” pane will show you key information about that destination, much of which is editable:



Of particular note:

- “ID” holds the unique identifier of this destination. This uses the same `@+id/` syntax as is used in layout and menu resources, though in the editor we drop off that prefix. So, while the editor may show `displayFragment`, the real identifier in the XML is `@+id/displayFragment`.
- “Class” is the Java/Kotlin class that implements this fragment or activity
- “Label” will be used in places where the Navigation component shows a user-readable name for the destination, such as in the app bar (e.g., a Toolbar)
- “Arguments” contains a list of the arguments that this particular destination expects to receive as input. The + button allows you to add a new argument:



In the preview area toolbar, you can click the “home” icon to designate the selected destination as being the start destination for this navigation graph.

Actions

Arrows connecting destinations in the preview area represent actions. Reminiscent of constraining widgets in a `ConstraintLayout`, you can drag a circle on the right edge of a destination to a corresponding circle on the left edge of another destination to create an action. This indicates that you want to be able to navigate from the first destination to the second.

NAVIGATING YOUR APP

Clicking on an arrow highlights it in blue and brings up its own set of attributes:

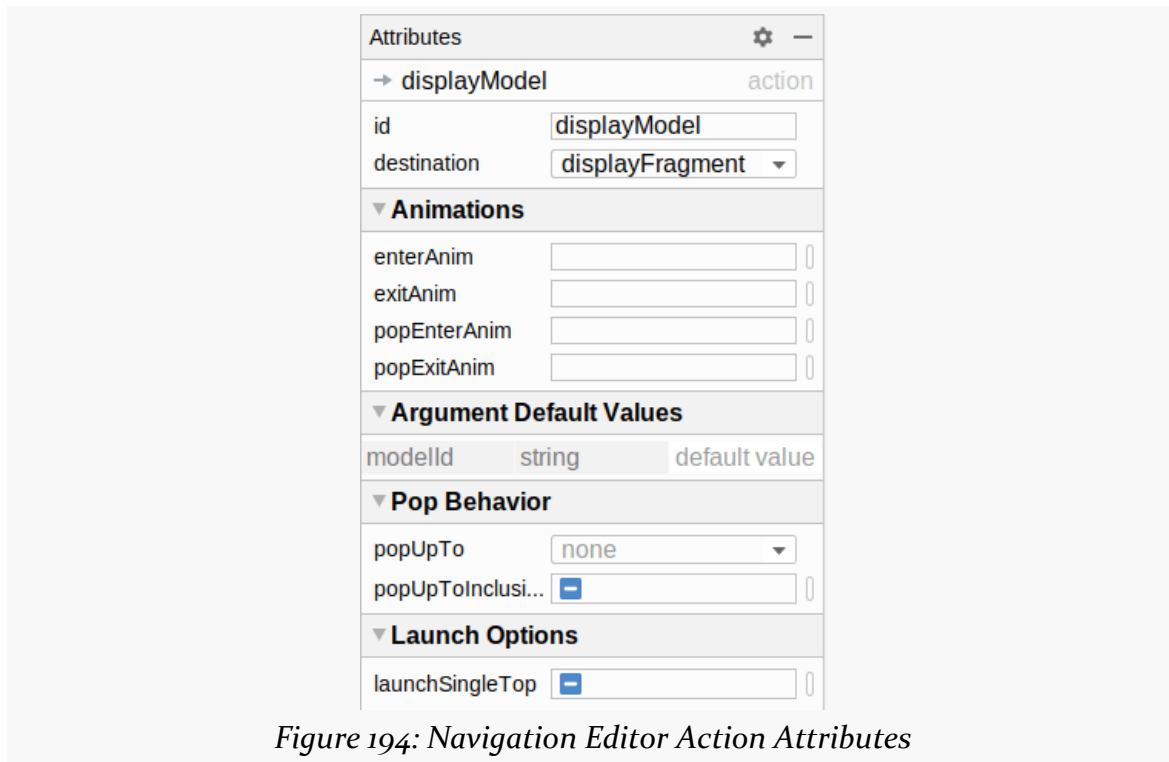


Figure 194: Navigation Editor Action Attributes

Of particular note:

- Actions have their own “ID” value, once again using the same @+id/ syntax that we have seen elsewhere
- “Destination” provides the ID of the destination that is the target of this action (i.e., the destination that will be navigated to if this action is performed)
- “Argument Default Values” allows you to optionally supply hard-coded values for arguments required by the chosen destination, instead of having to provide them in Java/Kotlin code

NavHostFragment

As noted earlier, the primary focus of the Navigation component is to navigate between fragments within an activity. This implies that the Navigation component knows where those fragments should go.

That is in the form of a NavHostFragment. This is a fragment that shows other

fragments, the ones specified in your navigation graph. You tell it what navigation graph it should use, and it will be responsible for:

- Showing your start destination, and
- Switching to another fragment destination, when your code asks for that at runtime

NavController

Switching to another destination in your navigation graph at runtime involves a `NavController`. This will be associated with the activity or fragment that is hosting the `NavHostFragment`. There are utility functions for getting the `NavController`, and from there you can ask it to move forwards or backwards in the navigation graph. “Forwards” means “go to this destination”; “backwards” means “go back to where we came from to get to our current destination”.

A Navigation-ized To-Do List

The `FragmentNav` sample module in the [Sampler](#) and [SamplerJ](#) projects mostly is a clone of [last chapter's](#) `FragmentManual` module. In this edition of the app, though, we use the `Navigation` component to get between the `ListFragment` and the `DisplayFragment`. We also switch to using a `Toolbar` here for our app bar, so we can get the up button added for us when we are on the `DisplayFragment`.

The Dependencies

The `Navigation` component is relatively complex in terms of dependencies, as we have variations for Java and Kotlin plus additional setup for the `Safe Args` feature.

The Basics

There are two main dependencies that most apps will use to employ the `Navigation` component:

1. `androidx.navigation:navigation-fragment` provides the core support for navigation, particularly using fragments
2. `androidx.navigation:navigation-ui` offers the integration with the `Toolbar` and other forms of app bar

These dependencies are closely coupled and generally will need to have the same

NAVIGATING YOUR APP

version. So, the overall Sampler/SamplerJ projects define a `nav_version` constant that contains the version to use:

```
buildscript {
    ext.nav_version = "2.3.5"

    repositories {
        google()
        jcenter()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:7.0.2'
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
    }
}
```

(from [build.gradle](#))

That way, when we add the dependencies in our `FragmentNav` module, we can reference that `nav_version` constant and ensure that everything stays synchronized:

```
implementation "androidx.navigation:navigation-fragment:$nav_version"
implementation "androidx.navigation:navigation-ui:$nav_version"
```

(from [FragmentNav/build.gradle](#))

Groovy — the language used for these Gradle scripts — supports string interpolation, using the same basic syntax as does Kotlin (`$name` to add the value of `name` to the string). As a result, values like `androidx.navigation:navigation-fragment:$nav_version` get the `$nav_version` portion replaced by whatever the value of `nav_version` is.

The KTX Bits

The dependencies closure shown above is from the `SamplerJ` project. Both Java and Kotlin projects are welcome to use those dependencies. Kotlin projects, though, are more likely to use the Android KTX variants of the dependencies:

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

(from [FragmentNav/build.gradle](#))

These, like the rest of the Android KTX libraries, add some extension functions and similar Kotlin features that make using the Navigation component easier in Kotlin.

The Safe Args Code Generator

For basic use of the Navigation component, those are all that you need. If you want to use the Safe Args feature for type-safe data exchange as part of navigation, you have some additional setup to perform.

Let's review one of the earlier code snippets, where we defined `nav_version`:

```
buildscript {
    ext.nav_version = "2.3.5"

    repositories {
        google()
        jcenter()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:7.0.2'
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
    }
}
```

(from [build.gradle](#))

If you look closely, this also adds another entry to the `buildscript` set of dependencies. Specifically, this adds a Gradle plugin for Safe Args. dependencies in a module usually refer to code added to that module for runtime use; dependencies in the `buildscript` closure usually refer to Gradle plugins that will be used by modules.

Simply having the `classpath` entry is insufficient, though. Modules need to opt into specific plugins offered by the library indicated by the `classpath` entry. So, our Java modules' `build.gradle` files have *two* `apply plugin` statements at the top: one for Android apps and one for Safe Args:

```
apply plugin: 'com.android.application'
apply plugin: 'androidx.navigation.safeargs'
```

(from [FragmentNav/build.gradle](#))

Kotlin modules will have additional plugins for Kotlin support, along with a Kotlin-specific version of the Safe Args plugin:

NAVIGATING YOUR APP

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-parcelize'
apply plugin: 'androidx.navigation.safeargs.kotlin'
apply plugin: 'kotlin-kapt'
```

(from [FragmentNav/build.gradle](#))

The Navigation Resource

The module has a `res/navigation/` directory for navigation resources. In there, there is a `main_nav_graph.xml` resource that represents our tiny navigation graph:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation android:id="@+id/main_nav_graph"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    app:startDestination="@id/listFragment">

    <fragment
        android:id="@+id/listFragment"
        android:name="com.commonware.jetpack.sampler.nav.ListFragment"
        android:label="ToDo Items">
        <action
            android:id="@+id/displayModel"
            app:destination="@id/displayFragment" />
        </fragment>
    <fragment
        android:id="@+id/displayFragment"
        android:name="com.commonware.jetpack.sampler.nav.DisplayFragment"
        android:label="ToDo Item">
        <argument
            android:name="modelId"
            app:argType="string" />
        </fragment>
    </navigation>
```

(from [FragmentNav/src/main/res/navigation/main_nav_graph.xml](#))

NAVIGATING YOUR APP

listFragment

There are two `<fragment>` elements set up as destinations in this resource. One is `listFragment`, pointing to our `ListFragment` class:

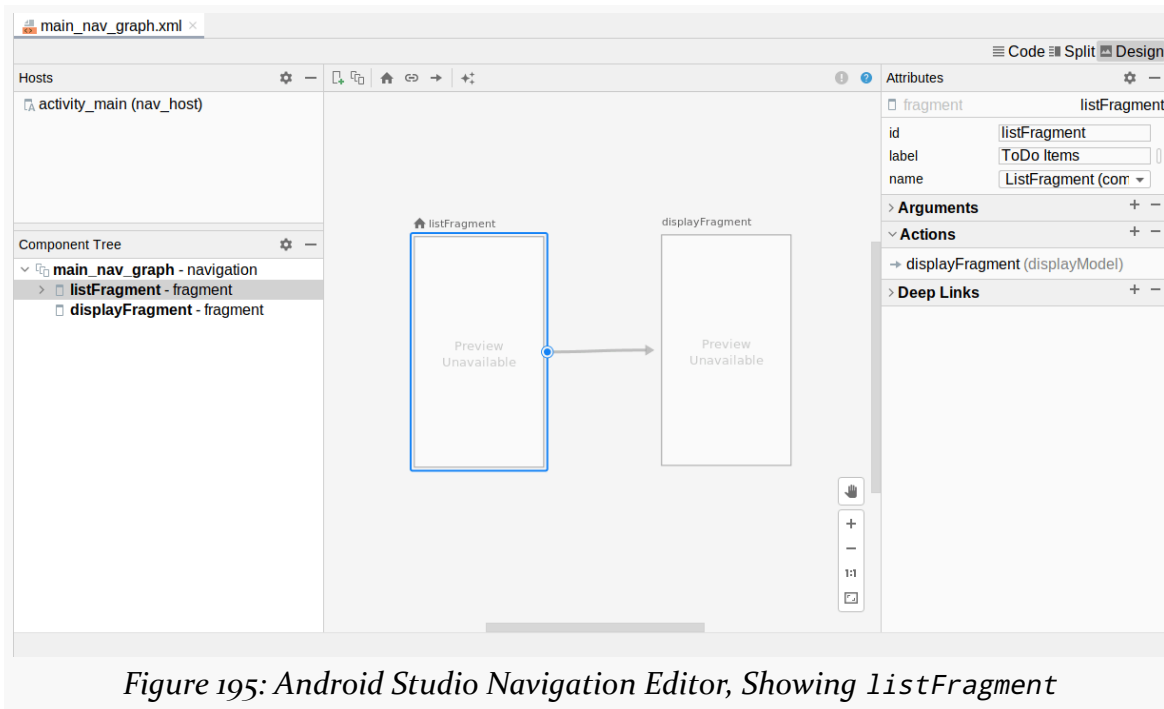


Figure 195: Android Studio Navigation Editor, Showing `listFragment`

It contains an `<action>` child element, identifying that this destination can navigate to the `displayFragment` destination. The action's ID is `displayModel`, and that is what we can use in our Java/Kotlin code to request to perform this action.

NAVIGATING YOUR APP

Clicking the action's arrow in the graphic designer shows the details of that action in the “Attributes” pane:

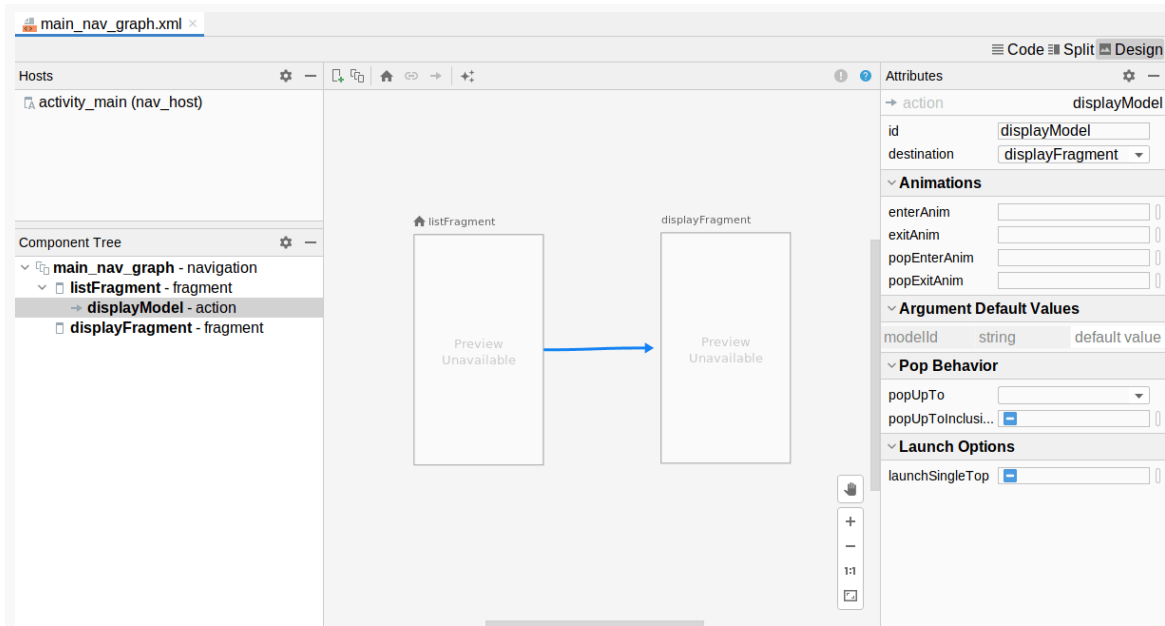


Figure 196: Android Studio Navigation Editor, Showing displayModel

The `app:startDestination` attribute in the root `<navigation>` element points to this `listFragment` destination, meaning that when we first use this navigation graph, the `ListFragment` is what should be displayed.

displayFragment

The other `<fragment>` element is for `displayFragment`:

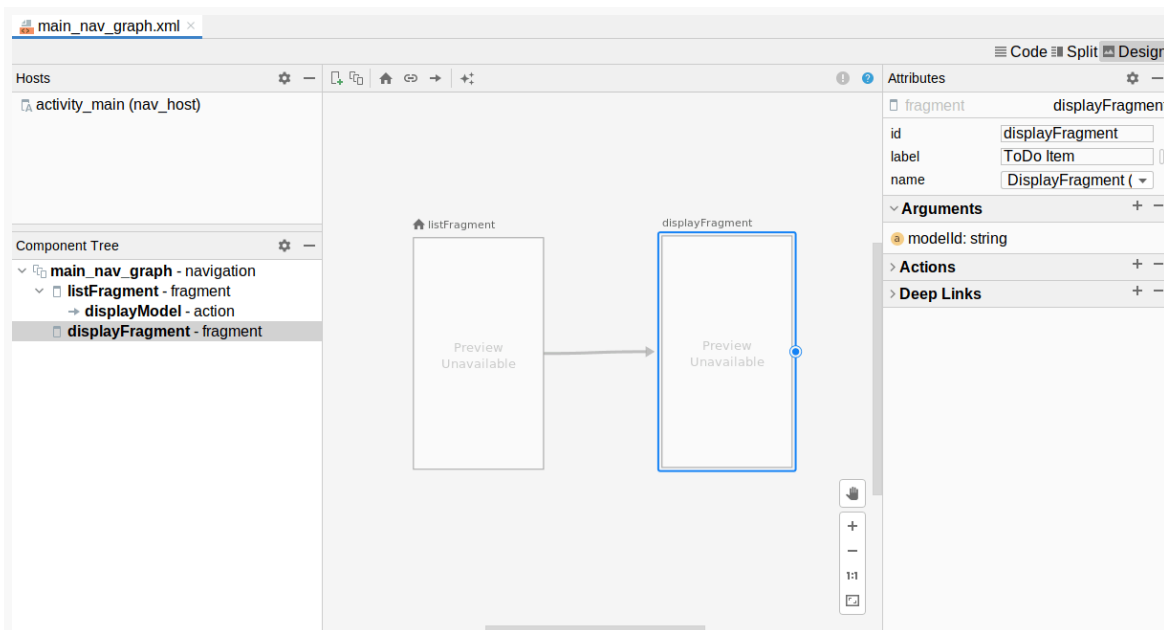


Figure 197: Android Studio Navigation Editor, Showing `displayFragment`

This contains an `<argument>` child element, stating that this destination expects to receive a `modelId` String value.

The Activity Layout

In the `FragmentManager` sample, we did not need an activity layout. We just used the existing `android.R.id.content` container as a target for our `FragmentManager` requests.

With the Navigation component, usually you will need a layout resource, particularly one where you place the `NavHostFragment` to say where the navigation graph's `<fragment>` destinations should appear. So, `FragmentManager` has an `activity_main` layout resource that does just that:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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=".MainActivity">

<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:theme="?attr/actionBarPopupTheme" />

<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/toolbar"
    app:navGraph="@navigation/main_nav_graph" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [FragmentNav/src/main/res/layout/activity_main.xml](#))

The `NavHostFragment` is added using a `<FragmentContainerView>` element in the layout XML. This is the modern way to add “static fragments”, ones that will be used all the time and cannot themselves be removed. The `android:name` attribute identifies the particular `Fragment` subclass to use, using the fully-qualified class name. In this case, we are not using one of our fragments, but instead are using `NavHostFragment` (or, more formally, `androidx.navigation.fragment.NavHostFragment`).

The `NavHostFragment` takes two main custom attributes in addition to the standard ones for sizing and positioning the fragment. `app:navGraph` supplies a reference to the navigation graph resource that this host will be using for its contents. `app:defaultNavHost` basically says “have the system’s BACK button route through this fragment’s navigation graph”, so BACK presses navigate backwards through the graph.

The layout also has a Toolbar to use as our app bar, with the NavHostFragment taking up the rest of the available space. As a result, this app uses Theme.AppCompat.Light.NoActionBar as a basis for its AppTheme style resource, to suppress the default action bar implementation, as we saw back in [the chapter on the app bar](#).

MainActivity

Most of the FragmentNav app's code is the same as its FragmentManual counterpart. We still have two fragments, each with its own viewmodel, with ToDoModel objects coming from a ToDoRepository. We also still have the same RecyclerView pieces for populating the list.

What differs is how our activity sets things up, how we navigate from the ListFragment to the DisplayFragment, and how we pass the ToDoModel ID between those fragments.

The FragmentManual edition of MainActivity conditionally executed a FragmentTransaction to bring up the ListFragment. The Navigation-enhanced MainActivity instead needs to initialize the Navigation component.

The Android KTX extensions for Kotlin cause the Java and Kotlin code to look somewhat different, even though in the end they do the same work:

- Obtain the NavController associated with our NavHostFragment
- Hook up that NavController to our Toolbar, to automatically manage things like the Toolbar title and up button

Java

```
package com.commonware.jetpack.samplerj.nav;

import android.os.Bundle;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;
import androidx.navigation.NavController;
import androidx.navigation.fragment.NavHostFragment;
import androidx.navigation.ui.AppBarConfiguration;
import androidx.navigation.ui.NavigationUI;

public class MainActivity extends AppCompatActivity {
    @Override
```

NAVIGATING YOUR APP

```
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    NavController nav =
        ((NavHostFragment) getSupportFragmentManager()
            .findFragmentById(R.id.nav_host))
            .getNavController();
    AppBarConfiguration appBarCfg =
        new AppBarConfiguration.Builder(nav.getGraph()).build();

    NavigationUI.setupWithNavController(findViewById(R.id.toolbar), nav,
        appBarCfg);
}
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/samplerj/nav/MainActivity.java](#))

In Java, to obtain the NavController, we:

- Get a FragmentManager via `getSupportFragmentManager()`
- Get the NavHostFragment from that FragmentManager via `findFragmentById()`
- Call `getNavController()` on the NavHostFragment

After that, we:

- Ask the NavController for the NavGraph object that represents the parsed navigation resource (`getGraph()`)
- Pass that to an `AppBarConfiguration.Builder` and use that to build an `AppBarConfiguration`
- Pass that along with the NavController and the Toolbar to `NavigationUI.setupWithNavController()`, to have the Navigation component automatically manage the contents of the Toolbar

Now, the Navigation component will fill in the Toolbar title from our destination label and handle the up button, as we navigate through the graph that we have set up for this NavHostFragment.

Kotlin

The Kotlin code does the same work, with less actual code:

NAVIGATING YOUR APP

```
package com.commonware.jetpack.sampler.nav

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.navigation.fragment.findNavController
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.setupWithNavController
import com.commonware.jetpack.sampler.nav.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)

        supportFragmentManager.findFragmentById(R.id.nav_host)?.findNavController()?.let { nav ->
            binding.toolbar.setupWithNavController(nav, AppBarConfiguration(nav.graph))
        }
    }
}
```

(from [FragmentNav/src/main/java/com/commonware/jetpack/sampler/nav/MainActivity.kt](#))

Android KTX gives us:

- A `findNavController()` extension function for Activity and Fragment, so we can get our NavController just by calling `findNavController()` and providing the ID of the NavHostFragment;
- A `setupWithNavController()` extension function on Toolbar, so we can add Navigation support for it more directly; and
- A global `AppBarConfiguration()` function that *looks* like a constructor, though it actually uses `AppBarConfiguration.Builder` “under the covers”

ListFragment

The change to ListFragment is in the `navTo()` function, as now we use the Navigation component instead of a `FragmentTransaction` to bring up the `DetailFragment`.

This app uses Safe Args, courtesy of the `<argument>` element in the navigation graph and the `androidx.navigation.safeargs` Gradle plugin. Hence, we get a `ListFragmentDirections` class code-generated for us. This class is named based on the initial destination (ListFragment) and has static methods for each of our actions that collect the arguments that we need to pass to the destination that we want to navigate to. Our action has `displayModel` as its ID, so `ListFragmentDirections` has a `displayModel()` method that we can call, either

NAVIGATING YOUR APP

from Java:

```
private void navTo(TodoModel model) {  
    NavHostFragment.findNavController(this)  
        .navigate(ListFragmentDirections.displayModel(model.id));  
}
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/samplerj/nav/ListFragment.java](#))

...or from Kotlin:

```
private fun navTo(model: TodoModel) {  
    findNavController().navigate(ListFragmentDirections.displayModel(model.id))  
}
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/ListFragment.kt](#))

To navigate to a destination, we:

- Call `displayModel()`, passing it our `TodoModel` ID as the argument, which gives us a `NavDirections` object that encapsulates our action ID (`displayModel`) and arguments
- Get our `NavController` using either `NavHostFragment.findNavController()` (in Java) or the `findNavController()` extension function in Kotlin
- Pass the `NavDirections` to the `navigate()` function on our `NavController`

The Navigation component will then do whatever is necessary to take us to that destination, updating the Toolbar as needed.

DisplayFragment

The only change in `DisplayFragment` is how we get our `TodoModel` ID. In `FragmentManual`, we used the arguments `Bundle` directly. In `FragmentNav`, we are using `Safe Args`, which gives us a code-generated `DisplayFragmentArgs` class. This class is named after our destination (`DisplayFragment`), knows how to obtain its arguments, and supplies them to us in type-safe functions, such as `getModelId()` for our `modelId` argument.

In Java, we get a `DisplayFragmentArgs` by calling the static `fromBundle()` method, providing it our arguments `Bundle`:

```
String modelId = DisplayFragmentArgs.fromBundle(getArguments()).getModelId();
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/samplerj/nav/DisplayFragment.java](#))

From there, a call to `getModelId()` returns the `ToDoModel` ID that we supplied to `ListFragmentDirections.displayModel()`.

In Kotlin, while you could do the same thing, you also have the `navArgs()` property delegate from Android KTX:

```
private val args: DisplayFragmentArgs by navArgs()
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/DisplayFragment.kt](#))

That knows how to get our arguments `Bundle` and set up our `DisplayFragmentArgs` instance, so we can just refer to its `modelId` property to get the `ToDoModel` ID:

```
val model = vm.getModel(args.modelId)
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/DisplayFragment.kt](#))

So... Was It Worth It?

The Navigation component, particularly with the Safe Args plugin, has some nice features and characteristics. If your app has a fairly simple navigation graph, and the UI is not too elaborate, the Navigation component may well be worth using.

However, every time you choose a framework like the Navigation component, you sacrifice flexibility in favor of ease of development. There will be certain types of navigation or UI structures that the Navigation component simply will not support. If you need those things, you may not be able to use the Navigation component for some or all of your app.

The Next Wave: Kotlin DSL

The Navigation component started out focused on navigation XML resources. Navigation has now evolved into three supported ways of defining a navigation graph:

1. navigation resources
2. A Kotlin domain-specific language (DSL) for use with fragments and similar destinations
3. A related Kotlin DSL for use with Jetpack Compose and composables

So, let's take a look at what the Kotlin DSL looks and works like.

What We Gain, What We Lose

On the plus side, using a Kotlin DSL means that you do not have to fuss with Yet Another Resource Type. Some developers love the separation imposed by the split between resources and programming languages; other developers dislike it. If you fall in the latter camp, the Kotlin DSL is likely to be something that you prefer.

Also, it is far simpler for the team behind the Navigation component to extend the DSL, as that is just standard Kotlin code from a Kotlin library. So, it is easier for the Navigation team to add new features or fix awkward APIs.

However, what we lose is tooling:

- There is no drag-and-drop editor for the Kotlin DSL — you are writing Kotlin code the same way as you would any other Kotlin code
- There is no support for automatic generation of “SafeArgs”-style code, though creating your own is not that difficult
- The original Kotlin DSL, that we will explore here, requires the use of unique numeric identifiers, requiring you to write a bit of extra code to define what those identifiers are

Defining the Identifiers

The `FragmentNavDSL` sample module in the [Sampler](#) contains a clone of `FragmentNav` that uses the Kotlin DSL to define the nav graph. There is no `SamplerJ` edition, as the Kotlin DSL requires Kotlin to be practical.

We need unique `Int` identifiers for:

- The nav graph itself
- Each destination
- Each action

We also need unique `String` identifiers for any arguments that we want to pass to a destination via an action.

To that end, we have a `NavGraph` object that holds those constants... and a little bit more:

```
package com.commonware.jetpack.sampler.nav

import androidx.core.os.bundleOf
import androidx.navigation.NavController

object NavGraph {
    const val Id = 1

    object Screen {
        const val List = 2
        const val Display = 3
    }

    object Action {
        object DisplayModel {
            const val Id = 4

            fun NavController.displayModel(modelId: String) {
                navigate(Id, bundleOf(Args.ModelId to modelId))
            }

            val DisplayFragment.modelId
                get() = arguments?.getString(Args.ModelId)
        }
    }

    object Args {
        const val ModelId = "ModelId"
    }
}
```

(from [FragmentNavDSL/src/main/java/com/commonware/jetpack/sampler/nav/NavGraph.kt](#))

We will look at those two extension functions defined in `DisplayModel` a bit later. The rest, though, are just a namespaced set of constants, following some organizing suggestions from Google.

Defining the Graph

In `FragmentNav`, our nav graph is set up as a navigation resource. Our `activity_main` layout refers to that nav graph from the `FragmentContainerView`. As a result, `MainActivity` gets the nav graph automatically and can just start configuring how Navigation works.

In `FragmentNavDSL`, we no longer have a navigation resource. While we still have the `FragmentContainerView`, it no longer has an `app:navGraph` attribute. Which

means that MainActivity has the honor and privilege of setting up the nav graph itself, using the Kotlin DSL:

```
package com.commonware.jetpack.sampler.nav

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.navigation.NavType
import androidx.navigation.createGraph
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.fragment
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.setupWithNavController
import com.commonware.jetpack.sampler.nav.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)

        supportFragmentManager.findFragmentById(R.id.nav_host)?.findNavController()
            ?.let { nav ->
                nav.graph = nav.createGraph(NavGraph.Id, NavGraph.Screen.List) {
                    fragment<ListFragment>(NavGraph.Screen.List) {
                        label = getString(R.string.list_title)

                        action(NavGraph.Action.DisplayModel.Id) {
                            destinationId = NavGraph.Screen.Display
                        }
                    }

                    fragment<DisplayFragment>(NavGraph.Screen.Display) {
                        label = getString(R.string.display_title)

                        argument(NavGraph.Args.ModelId) {
                            type = NavType.StringType
                        }
                    }
                }

                binding.toolbar.setupWithNavController(
                    nav,
                    AppBarConfiguration(nav.graph)
                )
            }
    }
}
```



```
    )  
    }  
}  
}
```

(from [FragmentNavDSL/src/main/java/com/commonsware/jetpack/sampler/nav/MainActivity.kt](#))

After obtaining our `NavController`, we use `createGraph()` on that `NavController` to build a `NavGraph` object that we assign to the `graph` property on the `NavController`. `createGraph()` is the root of the DSL — it takes a lambda expression or other function type that actually builds up the nav graph. The receiver (`this`) for that lambda expression is a `NavGraphBuilder`, which actually implements the functions that make up the DSL itself.

`createGraph()` takes two other parameters besides that lambda expression:

- The unique `Int` ID of this nav graph
- The ID of the root destination

That is equivalent to the `android:id` and `app:startDestination` attributes on the root `<navigation>` element of the navigation resource that we started with.

From there, we call `fragment()` twice. `fragment()` is the DSL equivalent of the `<fragment>` element in the navigation resource, defining a destination in the form of a fragment. `fragment()` takes a type declaration, which supplies the `Fragment` subclass that should be displayed when we navigate to this destination, filling the role of `android:name` in the resource. `fragment()` also takes the ID of the destination (equivalent to `android:id`) and another lambda expression or other function type to specify details of the destination. There, we set a `label` property (equivalent to `android:label`).

For the first `fragment()` call, we call `action()` to define an action that we can take from this fragment. `action()` serves as the `<action>` equivalent, and there we provide IDs for the action itself and the destination that this action should go to. The second `fragment()` call has an `argument()` nested DSL call, replacing the `<argument>` element, where we provide the name of the argument and its type.

After that, we can call `setupWithNavController()` as before to tie the `Toolbar` into the `Navigation` component.

On the whole, there is a fairly straightforward mapping of navigation resource contents into the Kotlin DSL, with functions and parameters taking the place of

elements and attributes.

Navigating Using the Graph

In the FragmentNav sample, to actually navigate, we used the NavController, coupled with the code-generated “directions” and “args” classes that the SafeArgs plugin created for us. Without SafeArgs, the Kotlin DSL dumps most of the problem of properly passing arguments on our lap.

But, we can make it appear a bit more seamless, using those extension functions in NavGraph:

```
fun NavController.displayModel(modelId: String) {
    navigate(Id, bundleOf(Args.ModelId to modelId))
}

val DisplayFragment.modelId
    get() = arguments?.getString(Args.ModelId)
```

(from [FragmentNavDSL/src/main/java/com/commonsware/jetpack/sampler/nav/NavGraph.kt](#))

To navigate from ListFragment to DisplayFragment, we had this in the FragmentNav sample:

```
private fun navTo(model: ToDoModel) {
    findNavController().navigate(ListFragmentDirections.displayModel(model.id))
}
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/ListFragment.kt](#))

That turns into:

```
private fun navTo(model: ToDoModel) {
    findNavController().displayModel(model.id)
}
```

(from [FragmentNavDSL/src/main/java/com/commonsware/jetpack/sampler/nav/ListFragment.kt](#))

displayModel() comes from NavGraph. Like displayModel() on ListFragmentDirections, displayModel() takes our ToDoModel ID as a parameter. The FragmentNavDSL edition of displayModel() wraps up that ID in a Bundle, keyed by our ModelId identifier, and passes that to navigate(). Effectively, that is all ListFragmentDirections.displayModel() is doing — we just had to write the code ourselves. And, along the way, by making displayModel() be an extension function

NAVIGATING YOUR APP

on NavController, the navTo() function becomes a bit less verbose.

On the DisplayFragment side, in FragmentNav, we could use the navArgs() property delegate and the generated DisplayFragmentArgs class:

```
private val args: DisplayFragmentArgs by navArgs()
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/DisplayFragment.kt](#))

We could then refer to a modelId property on args:

```
val model = vm.getModel(args.modelId)
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/DisplayFragment.kt](#))

That line turns into:

```
val model = modelId?.let { vm.getModel(it) }
```

(from [FragmentNavDSL/src/main/java/com/commonsware/jetpack/sampler/nav/DisplayFragment.kt](#))

modelId is now an extension property on DisplayFragment, defined in NavGraph. It gets the arguments Bundle and pulls out our ModelId argument. DisplayFragment can reference it the same as it would any other property. Once again, while we had to write the extension property ourselves, we can tweak it to make *using* that code a bit simpler than what we had with SafeArgs.

Dialogs

If you have used a desktop operating system written in the past 25 years or so, you are used to dialogs (otherwise referred to as “dialog boxes”, “modals”, and “those things that keep getting in the way of what I am trying to do”).

On the whole, mobile design tries to get away from dialogs. Where possible, try not to lock the user into doing one specific thing. So, for example, if you need the user to provide a username and password, rather than use a dialog, consider using a regular activity, or display a panel inside of an existing activity. That way, the user can not only give you those credentials, but the user can get to your help and support screens to learn more about *why* you are asking for those credentials.

However, Android does support dialogs, for those cases where you need them, and we will explore some options for those in this chapter.

A Tale of Four Dialogs

Android has a tendency of making simple things complicated, and dialogs are no exception. As a result, there are four main ways that you can set up a dialog: `Dialog`, `AlertDialog`, `DialogFragment`, and `Theme.AppCompat.Dialog`.

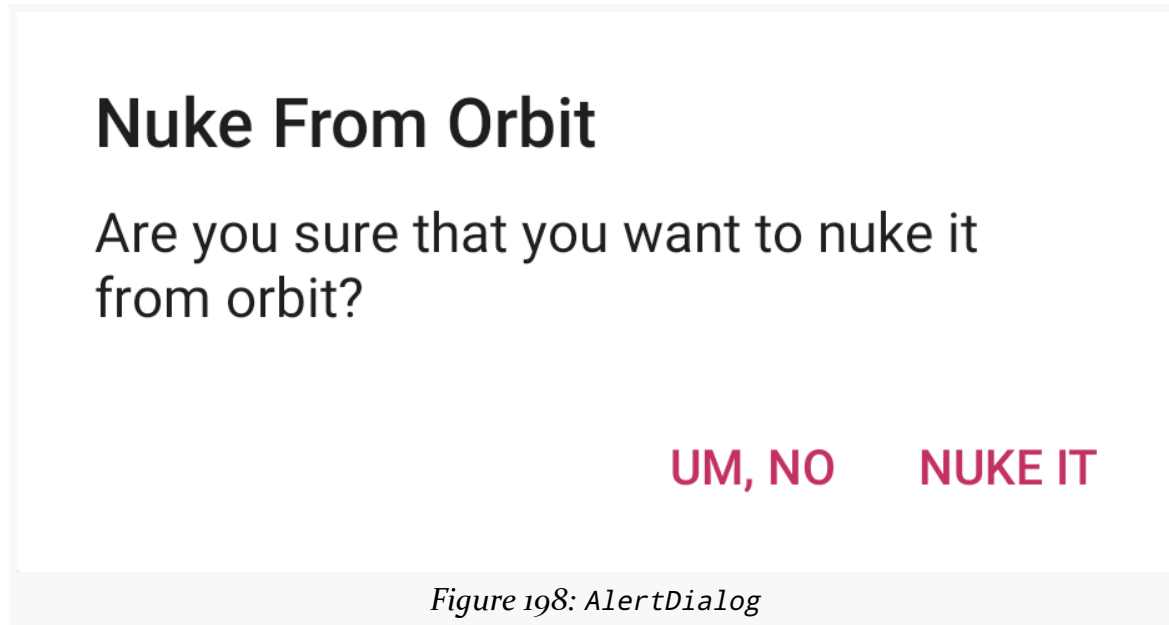
Dialog

The base for “true” dialogs in Android is `Dialog`. This class sets up a window that floats over top of your activity/fragment and displays whatever content you provide.

However, on its own, `Dialog` itself has no UI. It will display whatever you provide and does not “decorate” that with other elements.

AlertDialog

If you have used Android for a few years, you are probably used to seeing dialogs that are styled like this one:



This stock “look and feel” comes from AlertDialog. AlertDialog is a subclass of Dialog that offers a standard structure, including things like a title and positive (“Nuke It”) and negative (“Um, No”) buttons.

DialogFragment

A Dialog (or AlertDialog) is tied to the activity that displays it. If that activity is destroyed and recreated due to a [configuration change](#), a Dialog will *not* be re-displayed... at least, not without help.

DialogFragment is a wrapper around Dialog and AlertDialog that will re-display its dialog after a configuration change. Often times, we use DialogFragment so it handles that element of state for us.

There are two main ways of implementing a DialogFragment:

- If you override `onCreateView()`, like you would in a regular fragment, that UI will be wrapped in a Dialog and that Dialog is what is shown to the user

- Alternatively, if you override `onCreateDialog()`, you can return your own `Dialog`, such as an `AlertDialog`

We will see `DialogFragment` and `onCreateDialog()` shortly.

`Theme.AppCompat.Dialog`

Activities, by default, appear to fill the screen on mobile devices. In reality, they fill their window, and that window happens to mostly fill the screen on mobile devices most of the time. The story starts to get complicated when users go into split-screen mode, or for users using your app on desktop-style environments like Chrome OS, but the concept remains the same.

However, the reason why an activity fills its window, by default, is because that is what its theme says to do.

Not all themes do that. So, while `Theme.AppCompat` has its activity fill its window, `Theme.AppCompat.Dialog` does not. Instead, it only takes up whatever space is needed for its content, akin to how `wrap_content` works with layouts. That content is centered on the screen, and whatever activity is behind this one on the back stack is shown around the edges.

Visually, this appears like a dialog. So, if you find `DialogFragment` to be too constraining, you are welcome to use themes to have one or more activities appear like floating dialogs.

Using AlertDialog and DialogFragment

The NukeFromOrbit sample module in the [Sampler](#) and [SamplerJ](#) projects has a UI that consists of a really big button:

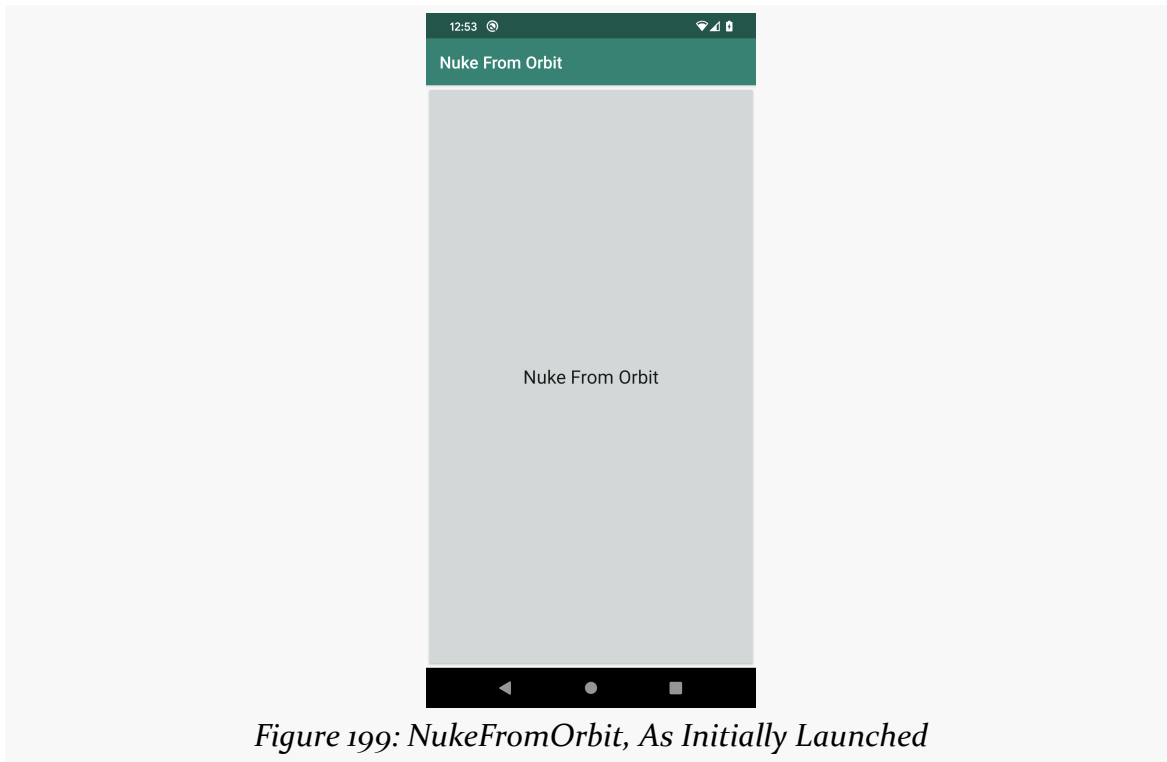


Figure 199: NukeFromOrbit, As Initially Launched

Clicking the button brings up the dialog shown earlier in the chapter:

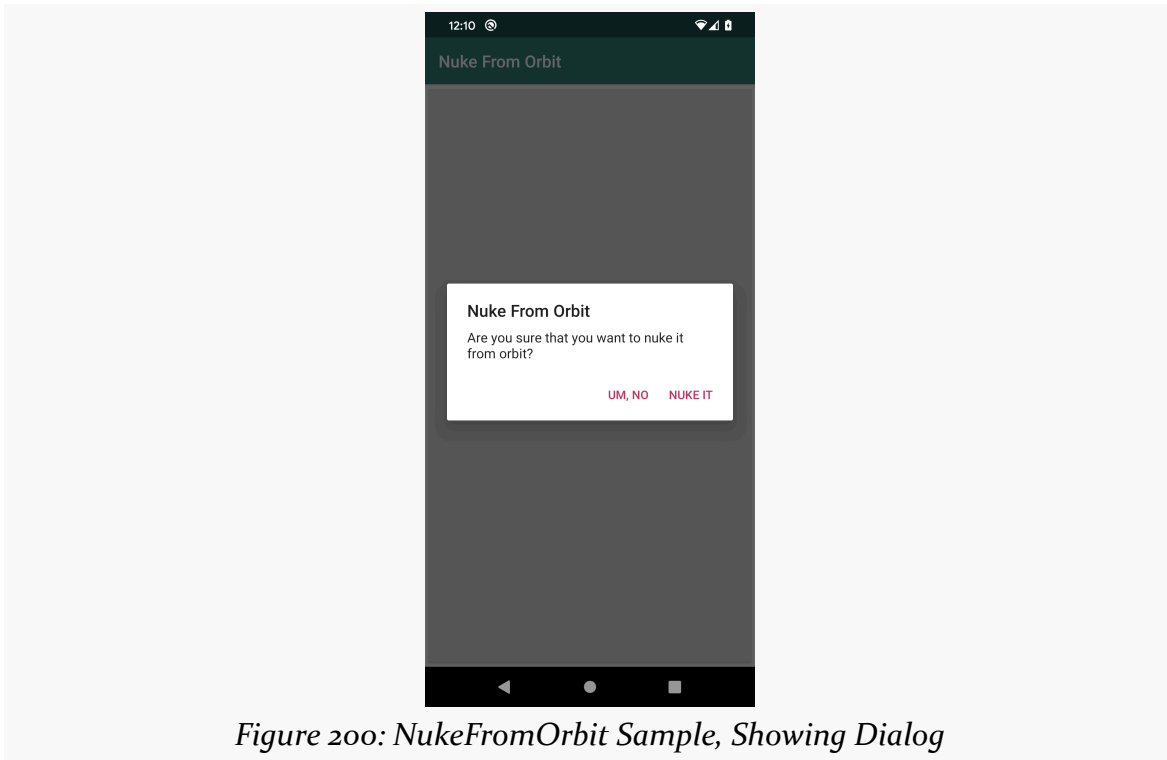


Figure 200: NukeFromOrbit Sample, Showing Dialog

Clicking either button dismisses the dialog, but if you click the “Nuke It” button, we also display a Toast.

Defining the Dialog

This particular sample uses `DialogFragment` and `AlertDialog` to display the fragment. The apps have a `ConfirmationDialogFragment` that overrides `onCreateDialog()` and creates an `AlertDialog` to display:

```
package com.commonware.jetpack.samplerj.dialog;

import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AlertDialog;
import androidx.fragment.app.DialogFragment;
import androidx.lifecycle.ViewModelProvider;
import androidx.navigation.NavController;
import androidx.navigation.fragment.NavHostFragment;
```


DIALOGS

```
public class ConfirmationDialogFragment extends DialogFragment {
    @NonNull
    @Override
    public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
        final NavController navController = NavHostFragment.findNavController(this);
        final ViewModelProvider viewModelProvider = new
ViewModelProvider(navController.getViewModelStoreOwner(R.id.nav_graph));
        final GraphViewModel vm = viewModelProvider.get(GraphViewModel.class);

        return new AlertDialog.Builder(requireActivity())
            .setTitle(R.string.dialog_title)
            .setMessage(R.string.dialog_message)
            .setPositiveButton(R.string.dialog_positive, (dialog, which) -> vm.onAccept())
            .setNegativeButton(R.string.dialog_negative, (dialog, which) -> vm.onDecline())
            .setOnCancelListener(dialog -> vm.onDecline())
            .create();
    }
}
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/samplerj/dialog/ConfirmationDialogFragment.java](#))

```
package com.commonsware.jetpack.sampler.dialog

import android.app.Dialog
import android.content.DialogInterface
import android.os.Bundle
import androidx.appcompat.app.AlertDialog
import androidx.fragment.app.DialogFragment
import androidx.lifecycle.ViewModelProvider
import androidx.navigation.fragment.NavHostFragment

class ConfirmationDialogFragment : DialogFragment() {
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        val navController = NavHostFragment.findNavController(this)
        val viewModelProvider =
            ViewModelProvider(navController.getViewModelStoreOwner(R.id.nav_graph))
        val vm = viewModelProvider.get(GraphViewModel::class.java)

        return AlertDialog.Builder(requireActivity())
            .setTitle(R.string.dialog_title)
            .setMessage(R.string.dialog_message)
            .setPositiveButton(R.string.dialog_positive) { _, _ -> vm.onAccept() }
            .setNegativeButton(R.string.dialog_negative) { _, _ -> vm.onDecline() }
            .setOnCancelListener { vm.onDecline() }
            .create()
    }
}
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/sampler/dialog/ConfirmationDialogFragment.kt](#))

To create an AlertDialog, we use an AlertDialog.Builder. This takes our activity as a constructor parameter, so we use requireActivity() to get the activity that is

hosting this fragment and use it. As the name suggests, `AlertDialog.Builder` offers a builder-style API to create the dialog, where we can call a series of functions one after the next. Specifically we configure:

- The title, which is displayed towards the top (“Nuke From Orbit”)
- The message, which is the simplest form of “content” for the dialog, consisting of just a piece of text to display (“Are you sure that you want to nuke it from orbit?”)
- The positive button (“Nuke It”) and negative button (“Um, No”)
- A listener for if the user cancels the dialog by other means, such as pressing the system BACK button

For the buttons, in addition to the caption, we also provide a lambda expression that will be invoked if the button gets clicked. What that lambda expression does, and the rest of the stuff in `onCreateDialog()`, we will explore more [later on](#).

Displaying the Dialog

The classic way of displaying a `DialogFragment` is to create an instance and call `show()` on it. This certainly works and you are welcome to use it.

However, if you are using the Navigation component, you can set up a `DialogFragment` as a destination in a navigation graph, simply by using a `<dialog>` element instead of a `<fragment>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph"
    app:startDestination="@id/mainFragment">

    <fragment
        android:id="@+id/mainFragment"
        android:name="com.commonware.jetpack.sampler.dialog.MainFragment"
        android:label="MainFragment" >
        <action
            android:id="@+id/confirmNuke"
            app:destination="@id/confirmationDialogFragment" />
        </fragment>
    <dialog
        android:id="@+id/confirmationDialogFragment"
        android:name="com.commonware.jetpack.sampler.dialog.ConfirmationDialogFragment"
        android:label="ConfirmationDialogFragment" />
</navigation>
```

(from [NukeFromOrbit/src/main/res/navigation/nav_graph.xml](#))

Here, we have `ConfirmationDialogFragment` set up in a graph, reachable by a

confirmNuke action from MainFragment.

MainFragment — which happens to be displaying that big button — can then just navigate to ConfirmationDialogFragment without knowing or caring that this is a dialog versus some other type of destination:

```
binding.nuke.setOnClickListener(v ->
    NavHostFragment.findNavController(MainFragment.this)
        .navigate(R.id.confirmNuke));
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/samplerj/dialog/MainFragment.java](#))

```
requireBinding().nuke.setOnClickListener {
    NavHostFragment.findNavController(this@MainFragment)
        .navigate(R.id.confirmNuke)
}
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/sampler/dialog/MainFragment.kt](#))

Here, we are using view binding to bind a click listener to the nuke button, so when the button is clicked, we show the dialog.

Sharing a ViewModel

Often, an activity or fragment needs a private ViewModel for managing its own data. However, sometimes, it would be useful to have shared ViewModel objects. For example, in this sample app, MainFragment needs to know whether the user clicked the positive or the negative button. This implies some amount of data-sharing between MainFragment and ConfirmationDialogFragment, and sharing a ViewModel would be one approach for that sort of sharing.

That is handled by having the right ViewModelProvider:

- An activity that creates a ViewModelProvider via `ViewModelProvider(this)` will get ViewModel objects tied to the activity
- A fragment that creates a ViewModelProvider via `ViewModelProvider(this)` will get ViewModel objects tied to the fragment
- A fragment that creates a ViewModelProvider by passing *the hosting activity* to the ViewModelProvider constructor will get ViewModel objects tied to the activity... and if the activity uses `ViewModelProvider(this)`, the activity and the fragment will share those ViewModel objects

In Kotlin, instead of using the `viewModels()` property delegate, you can use

`activityViewModels()` to get a `ViewModel` from the activity's scope, sharing that `ViewModel` with the activity.

It is also possible to get a `ViewModelProvider` tied not to an activity or a fragment, but to a navigation graph. All of the destinations in that navigation graph will share `ViewModel` objects obtained from the graph-specific `ViewModelProvider`.

To do that, we:

- Obtain a `NavController` for the fragment, via `NavHostFragment.findNavController()`
- Create a `ViewModelProvider`, passing in a `ViewModelStoreOwner` obtained from the `NavController`
- Use that `ViewModelProvider` for `ViewModel` objects to be shared among destinations in the navigation graph

Both `MainFragment` and `ConfirmationDialogFragment` do this via similar blocks of code:

```
final NavController navController = NavHostFragment.findNavController(this);
final ViewModelProvider viewModelProvider = new
ViewModelProvider(navController.getViewModelStoreOwner(R.id.nav_graph));
final GraphViewModel vm = viewModelProvider.get(GraphViewModel.class);
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/samplerj/dialog/ConfirmationDialogFragment.java](#))

```
val navController = NavHostFragment.findNavController(this)
val viewModelProvider =
    ViewModelProvider(navController.getViewModelStoreOwner(R.id.nav_graph))
val vm = viewModelProvider.get(GraphViewModel::class.java)
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/sampler/dialog/ConfirmationDialogFragment.kt](#))

Both our `MainFragment` instance and our `ConfirmationDialogFragment` instance now share a `GraphViewModel` instance.

Reacting to the Dialog

But what we really want is to get the button-click events from `ConfirmationDialogFragment` to `MainFragment`. The `GraphViewModel` is a key piece of that, but we need some other elements as well... and we will take a look at those in [an upcoming chapter](#).

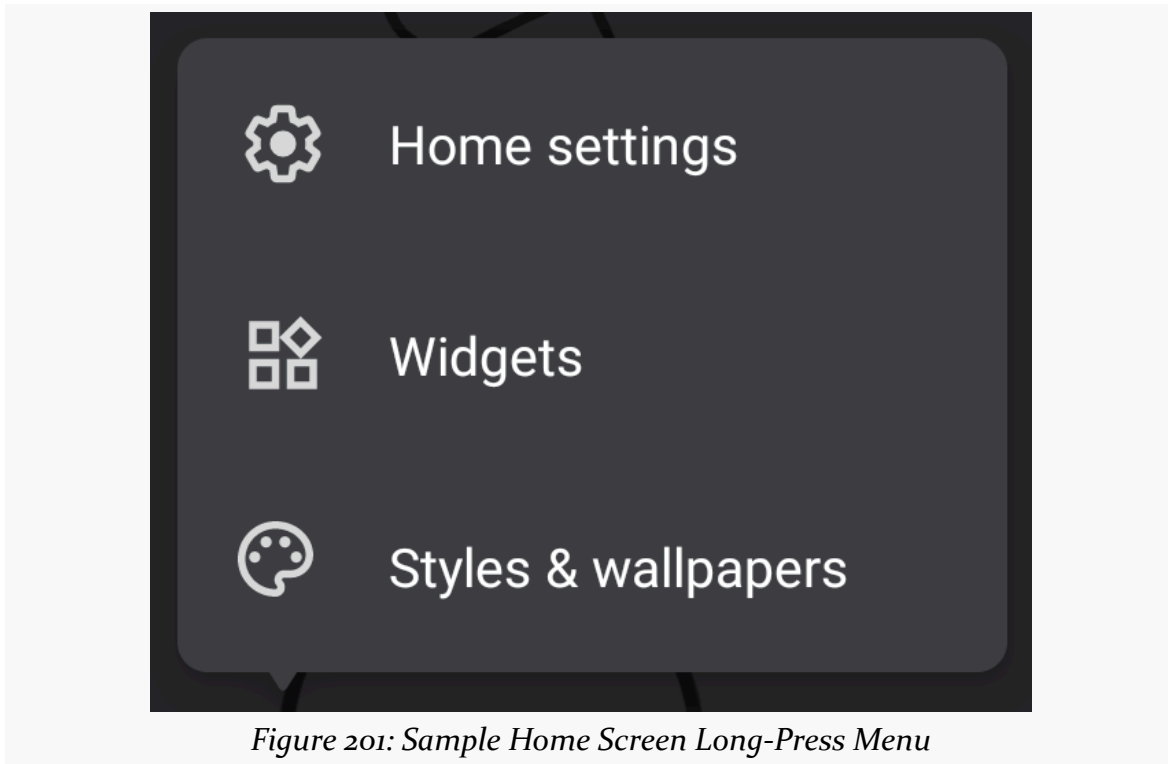
Writing an App Widget

In 2020, iOS finally [added support for app widgets](#), a capability that Android has had for over a decade. Since iOS tends to have a big influence in app design and feature decisions, interest in app widgets on Android is likely to increase. So, in this chapter, we will explore what it takes to write an app widget.

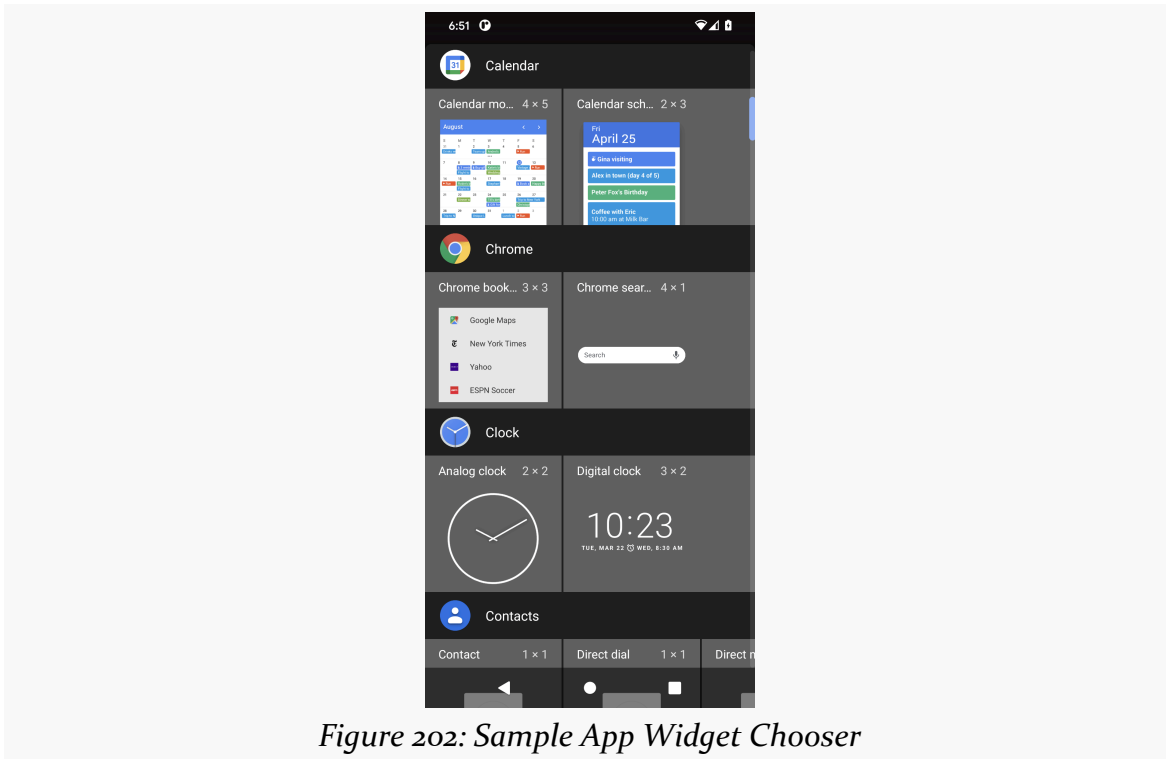
Writing a... What?

“App widgets” are interactive elements that can be added to a home screen. While some things that appear like app widgets are actually features of the launcher app, true app widgets are ones that can be published by apps such as yours.

Users can add app widgets to their home screen as they see fit, typically via a long-click on the wallpaper or home screen background.



Choosing “Widgets” from that particular home screen menu brings up a chooser of available app widgets:



The user can then drag and drop the app widget to the home screen. Usually, the user can long-press on the app widget to be able to resize or remove it from the home screen.

For most apps, the idea behind app widgets is to make key bits of information or key actions available to the user just from the home screen, without having to switch to your app’s full UI.

Challenges with App Widgets

Historically, relatively few apps publish app widgets. While app widgets are reasonably popular among “power users”, but they are not easy to discover. As noted above, iOS adding an equivalent capability will help drive user interest, and that in turn is likely to drive developer interest.

However, there is another reason why relatively few apps publish app widgets:

developing app widgets is **seriously** weird.

Somebody Else Draws the UI

Android app developers are used to drawing the app's UI themselves, typically in the form of widgets in activities and fragments as we have seen here in this book.

App widgets use the same sort of `View` and `ViewGroup` subclasses that we use for activities and fragments, with a lot of limitations. However, our apps do not actually draw those widgets. Instead, the home screen does. We tell the home screen “please have my app widget contain X”, and the home screen is responsible for actually doing the work.

Inter-Process Communication

To make that possible, we have to arrange to get our requested app widget UI over to the home screen for rendering. Since the home screen is part of the launcher app, it runs in a completely separate OS process from our own. So, our process needs to tell that other process what to do, and this makes things cumbersome.

Part of the problem is that for such inter-process communications to work, both sides have to agree on how to send and receive information. That makes changes to the communications protocol challenging: if one side sends something that the other side does not understand, then something may go wrong with the app widget.

Lots of Tumbleweeds

That protocol problem means that Google has not changed how app widgets work very much over the years. App widgets were introduced in Android 1.5, back when dinosaurs roamed the Earth.

(OK, perhaps not *that* long ago, but quite a while)

While app widgets got one round of improvements in Android 3.0 and a few minor tweaks since then, app widgets are still largely stuck with a protocol that is over a decade old.

Plus, because both sides have to agree on the protocol, we cannot use things like `RecyclerView` — while our app might have that, the launcher app might not.

As a result, app widgets have a very limited roster of available `View` subclasses that

you can use, and some of those subclasses are ones that you would not use normally because they have been replaced by newer alternatives.

Introducing Broadcasts and Receivers

To date, we have focused on one type of Android component: the activity. There are actually four different types of components. App widgets are based on another component: the `BroadcastReceiver`.

You might think that a `BroadcastReceiver` receives broadcasts. In truth, it does, but the story is a bit more complex than that.

In Android, we can “broadcast” an `Intent`. Instead of using `startActivity()`, we use `sendBroadcast()`. An `Intent` that is broadcast could be very generic, in which case it might be truly “broadcast” to a variety of apps on the device that are set to receive such a broadcast. Nowadays, more commonly, the `Intent` is an explicit `Intent`, identifying a particular component that should receive it. While technically this is still termed a “broadcast”, only that designated component will receive it, not any arbitrary app.

Normal Android user interfaces are created by activities, perhaps with the assistance of fragments. In contrast, app widget user interfaces are created using a `BroadcastReceiver`. Android will send a special broadcast to our `BroadcastReceiver`, telling us to provide the UI definition. That UI definition is then passed along to the home screen, which will render that UI for us.

Pieces of the App Widget

In some respects, setting up an app widget is like setting up an activity:

- You usually need a layout resource
- You need a manifest entry
- You need a Java or Kotlin class

The details, though, are substantially different between activities and app widgets.

The Layout

You will need a layout resource for your app widget. In principle, this works the same as any other layout resource.

However, you have a very limited palette of available widgets and containers to choose from:

- AdapterViewFlipper
- Button
- Chronometer
- FrameLayout
- ImageButton
- ImageView
- GridLayout
- GridView
- LinearLayout
- ListView
- ProgressBar
- RelativeLayout
- StackView
- TextView
- ViewFlipper

Of those, Button, FrameLayout, ImageView, ProgressBar, and TextView are still popular today. Most of the rest are either unpopular (ImageButton, StackView) or are largely replaced (e.g., RecyclerView replacing ListView and GridView). However, you may find yourself using some of those less-popular options, simply because you cannot use things like RecyclerView and ConstraintLayout in an app widget.

The AppWidgetProvider

Instead of an Activity subclass, you will create an AppWidgetProvider subclass. AppWidgetProvider in turn extends BroadcastReceiver.

While there are a variety of methods that you could implement, the big one will be `onUpdate()`. This will be called when the OS proactively wants an updated copy of the UI for the app widget.

The Metadata

An app widget also needs a resource that serves in the role of metadata, describing various aspects of the app widget's behavior. For example, you can provide details about the initial size of the app widget, or supply a preview image to show in the app widget selector UI.

While this resource is an XML file, it is not a layout, menu, or other “major” type of resource. Instead, it will go in a `res/xml/` directory of your source set — this is used for arbitrary XML documents. We will see it again later in the book when we look at [the preference system](#).

The Manifest Entry

There are a couple of ways to tell Android about a `BroadcastReceiver`. The approach used for app widgets is to have a `<receiver>` element in the manifest, much like how we have an `<activity>` element for our activities. So, the app widget will need a `<receiver>` pointing to the `AppWidgetProvider` subclass.

In addition, this `<receiver>` element will have a child XML element that tells Android about that metadata XML resource. This is a bit reminiscent of how our launcher activity has a child `<intent-filter>` element, just with a different syntax and role.

Colors in an App Widget

The `ColorWidget` sample module in the [Sampler](#) and [SamplerJ](#) projects mostly is a variant on the “show a list of colors” samples that we have seen already. This one, though, also has an app widget, to show a random color to the user. It also:

- Offers a refresh button, in case you do not like that color
- Shows the `BigSwatchFragment` for the random color if the user taps on the app widget (other than on the refresh button)

The Layout

The layout resource describing the UI of the app widget (`res/layout/widget.xml`) is fairly conventional... except for the choice of container:

```
<?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:id="@+id/root"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:padding="8dp"
```

```
tools:background="#FFFF0000">

<TextView
    android:id="@+id/label"
    style="?android:attr/textAppearanceLarge"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_weight="1"
    android:typeface="monospace"
    tools:text="#abcdef" />

<ImageButton
    android:id="@+id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:contentDescription="@string/menu_refresh"
    android:src="@drawable/ic_refresh_black_24dp" />

</LinearLayout>
```

(from [ColorWidget/src/main/res/layout/widget.xml](#))

Here, we wrap a `TextView` and an `ImageButton` in a `LinearLayout`. `LinearLayout` was mentioned briefly back in [the chapter on ConstraintLayout](#), where `LinearLayout` was mentioned as a “legacy” option. If we were creating an activity, fragment, or `RecyclerView` row here, we could have used `ConstraintLayout` for the container. That is not an option with an app widget, as `ConstraintLayout` is not on [the list of eligible widgets and containers](#).

`LinearLayout` positions its children one after the next in a row or column. The `android:orientation` attribute will be `horizontal` for a row and `vertical` for a column. Here we have it set to be `horizontal`.

The `TextView` will appear first in the row, followed by the `ImageButton`. The `ImageButton` has its natural width (`wrap_content`). The `TextView` has a width of `0dp` and a `layout_weight` of `1`. The net effect is that the `ImageButton` will get its space in the `LinearLayout`, and the `TextView` will take over all the rest of the space. This shoves the `ImageView` to the far end side of the row.

Both the `TextView` and the `ImageButton` have `android:layout_gravity` set to `center_vertical`. `android:layout_gravity` is a hint to the `LinearLayout` as to where these widgets would like to be positioned, particularly in the opposite axis of the flow of the `LinearLayout`. `center_vertical` means “for this row, position these

widgets centered vertically within the available space”.

The Core AppWidgetProvider

The “brains” of an app widget comes in the form of your `AppWidgetProvider` subclass. In the sample project, that is named `ColorWidget`. Much of it is unlike anything else that you have done in Android app development... which is one of the reasons why many app developers do not fuss with creating app widgets.

Our primary entry point in an `AppWidgetProvider` is an `onUpdate()` function. This is a bit like `onCreate()` of an activity, in that it is where we set up our UI. It is called when we first need to set up that UI.

Creating the RemoteViews

As mentioned earlier, our app’s process does not render that UI. Instead, we need to define what the UI will contain, then arrange to get that UI over to the OS. The OS, in turn, will pass that UI definition to the launcher app, which can then render that UI.

As such, while we can work with layout resources, we do not use `LayoutInflater` or view binding or things like that to set up our UI definition. Instead, we use a `RemoteViews` object. As the name suggests, `RemoteViews` represents a UI definition designed to be passed to another “remote” app for rendering.

To create a `RemoteViews`, just create an instance, supplying our application ID and the resource ID of a layout resource to use:

```
RemoteViews remoteViews =  
    new RemoteViews(context.getPackageName(), R.layout.widget);
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
val remoteViews = RemoteViews(context.packageName, R.layout.widget)
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

Here, we reference `R.layout.widget`, pointing to the widget layout we examined earlier.

Setting the Color

Usually, our app widget has a dynamic UI, just as our activities and fragments do. While the overall structure is constant, elements of data change. In this case, we want our app widget to display the hex value of a random color, plus render the background of the widget itself in that color.

With activities and fragments, we would call functions on the widgets and containers to make these sorts of changes. In this case, we do not have those widgets — all we have is the RemoteViews. So, we call functions on the RemoteViews to affect these changes, with parameters identifying which widget or container is the one that we want to modify:

```
int color = random.nextInt();

remoteViews.setTextViewText(
    R.id.label,
    context.getString(R.string.label_template, color)
);
remoteViews.setInt(R.id.root, "setBackgroundColor", color);
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
val color = random.nextInt()

remoteViews.setTextViewText(
    R.id.label,
    context.getString(R.string.label_template, color)
)
remoteViews.setInt(R.id.root, "setBackgroundColor", color)
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

Here, we generate a random color (`color`), then make two calls to functions on RemoteViews. The first one is easy: `setTextViewText()` is the RemoteViews equivalent of calling `setText()` on a `TextView`. We provide the ID of the `TextView` widget in our layout along with the text that we want to fill in.

However, very few functions exist on RemoteViews for manipulating the contents of the layout. For example, there is no `setBackgroundColor()` function to set the background color of some widget or container.

There is a family of functions that allow us to invoke arbitrary View functions. One is `setInt()`. We provide the ID of the widget to manipulate, the actual function that

we want to have invoked, and an `Int` value. This information is tucked into the `RemoteViews`, and the launcher app will arrange to call this function for us.

Unfortunately, there are two problems with this approach:

- Not every `View` function can be called this way
- Which ones can and cannot be called this way is undocumented.

The only way to know is to look at the source code to `View` and see if the function has the `RemoteableViewMethod` annotation on it. `setBackgroundColor()` does:

```
/**
 * Sets the background color for this view.
 * @param color the color of the background
 */
@RemoteableViewMethod
public void setBackgroundColor(@ColorInt int color) {
    if (mBackground instanceof ColorDrawable) {
        ((ColorDrawable) mBackground.mutate()).setColor(color);
        computeOpaqueFlags();
        mBackgroundResource = 0;
    } else {
        setBackground(new ColorDrawable(color));
    }
}
```

There are other things that we are configuring on the `RemoteViews`, but we will explore them later in the chapter.

Updating the App Widget

While `onUpdate()` is our entry point, that is not where the code shown above appears. That code is in a private `updateWidget()` function that we call from `onUpdate()`:

```
@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager,
                    int[] appWidgetIds) {
    for (int appWidgetId : appWidgetIds) {
        updateWidget(context, appWidgetManager, appWidgetId);
    }
}
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

WRITING AN APP WIDGET

```
override fun onUpdate(  
    context: Context,  
    appWidgetManager: AppWidgetManager,  
    appWidgetIds: IntArray  
) {  
    appWidgetIds.forEach { updateWidget(context, appWidgetManager, it) }  
}
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

The user can not only add your app widget to their home screen, but they can do so *multiple times*. Each app widget is identified by an integer ID value. `onUpdate()` is passed an array of IDs, indicating all of the app widgets that need to be updated. Our `onUpdate()` functions iterate over the array and call `updateWidget()` for each, supplying that app widget ID to `updateWidget()`, along with a `Context` and an `AppWidgetManager`.

In `updateWidget()`, towards the bottom, we use `AppWidgetManager` to tell Android about our new `RemoteViews` for this app widget:

```
appWidgetManager.updateAppWidget(appWidgetId, remoteViews);
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
appWidgetManager.updateAppWidget(appWidgetId, remoteViews)
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

`updateAppWidget()` on `AppWidgetManager` takes the app widget ID and the `RemoteViews` and arranges to pass those to the launcher, so it can update the associated UI on the home screen.

For many app widgets, you will have data in files, databases, Web services, etc. that are tied to the app widget. You can use the app widget ID to help distinguish one app widget from another, if they each need to have separate data.

The Metadata

We also have the metadata XML resource, in `res/xml/widget_metadata.xml`:

```
<?xml version="1.0" encoding="utf-8"?>  
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"  
    android:minWidth="250dp"  
    android:minHeight="40dp"  
    android:previewImage="@drawable/widget_preview" />
```

(from [ColorWidget/src/main/res/xml/widget_metadata.xml](#))

The root — and only — element is `<appwidget-provider>`. Its attributes describe the nature of the app widget, in particular to help the home screen understand how to use it.

Specifically, here we use:

- `android:minWidth` and `android:minHeight` to suggest a starting size for the app widget
- `android:previewImage` to supply a sample of what the app widget will look like, for use in the app widget selector UI

This is pretty much the bare minimum of what realistically you can provide for an app widget's metadata. There are many other options, such as describing how the app widget can be resized, whether its contents should be updated automatically, etc.

The Manifest Entry

Our `ColorWidget` also needs to be registered in the manifest as a `<receiver>`:

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

    <application
        android:allowBackup="false"
        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>
        <receiver android:name=".ColorWidget">
            <intent-filter>
                <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

WRITING AN APP WIDGET

```
<meta-data
    android:name="android.appwidget.provider"
    android:resource="@xml/widget_metadata" />
</receiver>
</application>

</manifest>
```

(from [ColorWidget/src/main/AndroidManifest.xml](#))

The `android:name` attribute works like its equivalent on `<activity>`, identifying the class that implements the receiver. In this case, `.ColorWidget` says to use the `ColorWidget` class inside of our package (`com.commonware.jetpack.sampler.widget`).

Just as having a MAIN/LAUNCHER `<intent-filter>` is a requirement of activities that should appear in the launcher, an APPWIDGET_UPDATE `<intent-filter>` is a requirement of app widgets. The OS will send a message to our `ColorWidget`, using that action, when it is time for us to either set up the app widget UI or refresh it. That, in turn, is what triggers a call to `onUpdate()` in our `ColorWidget`.

Finally, the `<meta-data>` element is a way that you can add configuration details to the manifest for arbitrary stuff that the build tools really know about. In this case, the OS is expecting us to point to that XML metadata resource that we created. The `android:name="android.appwidget.provider"` attribute says “this is what metadata we are providing” and `android:resource="@xml/widget_metadata"` attribute says “and here is where you can find the actual metadata”.

The Basic Results

If you run the app, then add the app widget to your home screen, you will see a color represented in the rectangular app widget background, plus the hex value of the color in text:

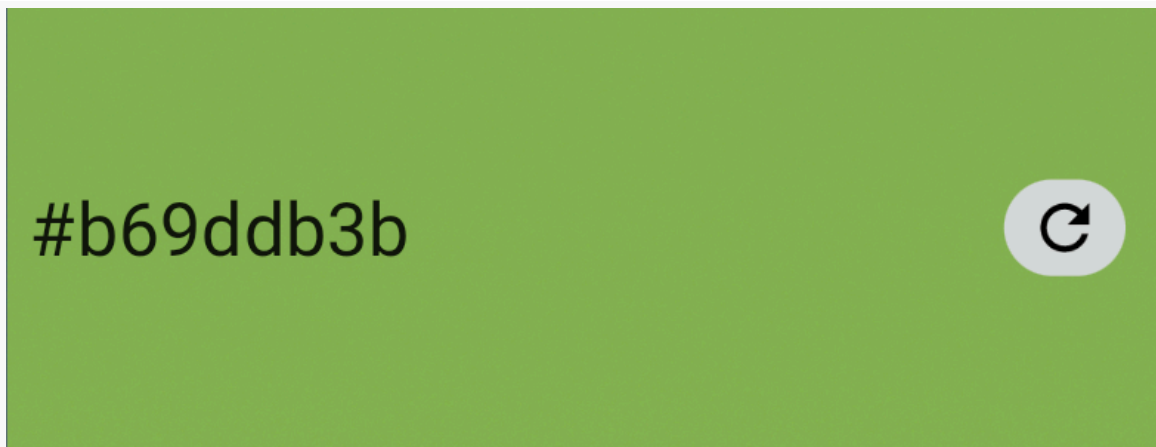


Figure 203: ColorWidget App Widget, Showing a Nice Shade of Green

Actions and App Widgets

App widgets can do more than render a bit of UI on the home screen. They can also respond to click events, either to update the contents of the app widget or to take some other action, such as displaying an activity.

However, the app widget UI is not part of the app's process. We do not have access to the actual UI objects, so we cannot call `setOnClickListener()` on them. Instead, we have to go through a bit of indirection, in the form of `PendingIntent` objects.

Introducing `PendingIntent`

We have seen `Intent` objects before. We use them to identify activities to start. As it turns out, we can also use `Intent` objects with two other types of components:

- We can start a service, where the `Intent` identifies the service to be started
- We can send a broadcast, where the `Intent` is the broadcast, identifying who should receive the message

A `PendingIntent` is a wrapper around an `Intent` that, among other things, indicates

what action should be used with that Intent, such as starting an activity or sending a broadcast. There is a *lot* more to a PendingIntent than that, particularly with respect to security. But, for the purposes of this chapter, considering it to be a combination of an Intent and a statement of what to do with the Intent will suffice.

A PendingIntent is a way of passing an action to be performed to some other app or to the OS. In the case of app widgets, we can register click “listeners” that, instead of invoking some listener object or lambda expression in our app, invokes a PendingIntent. So, we can tie clicks in our app widget to PendingIntent objects, which in turn tell Android to start an activity, send a broadcast, etc.

Refreshing the Color

With that in mind, let’s look at two actions that we want to perform from our app widget.

First, there is the refresh button on the one side of the app widget. When the user taps that button, we want to generate a new random color and update the UI of the app widget to show that color.

Since we want to update the app widget, one approach is to route control back to ColorWidget, since we already have the logic there to update the app widget UI.

Creating the PendingIntent

In `updateWidget()`, as part of configuring the RemoteViews, we create an Intent, identifying our ColorWidget:

```
Intent refreshIntent = new Intent(context, ColorWidget.class)
    .setAction(ACTION_REFRESH)
    .putExtra(EXTRA_APP_WIDGET_ID, appWidgetId);
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
val refreshIntent = Intent(context, ColorWidget::class.java)
    .setAction(ACTION_REFRESH)
    .putExtra(EXTRA_APP_WIDGET_ID, appWidgetId)
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

We created Intent objects for our own components before, when starting activities. We use the same Intent constructor here as then, to supply our ColorWidget class and indicate that is what should be started. We also do two other things as part of

configuring the Intent:

- We call `setAction()`, to associate a custom action string with our Intent. We can use this to distinguish cases where the `ColorWidget` is invoked based on our own Intent from cases where it is invoked for other reasons.
- We call `putExtra()` to attach the app widget ID to the Intent. This way, we know which app widget is the one that the user tapped the refresh button on.

Then, we wrap that Intent in a `PendingIntent`:

```
PendingIntent refreshPI = PendingIntent.getBroadcast(  
    context,  
    appWidgetId,  
    refreshIntent,  
    PendingIntent.FLAG_UPDATE_CURRENT  
);
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
val refreshPI = PendingIntent.getBroadcast(  
    context,  
    appWidgetId,  
    refreshIntent,  
    PendingIntent.FLAG_UPDATE_CURRENT  
)
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

The function on `PendingIntent` that we call is `getBroadcast()`, saying that we want the system to send a broadcast using our supplied Intent. We also provide:

- a Context
- a unique ID for this `PendingIntent` (in this case, we use the app widget ID)
- the Intent to be wrapped in the `PendingIntent`
- a flag, the details of which are well beyond the scope of this book

Connecting the `PendingIntent`

Given the `PendingIntent`, attaching it to a click listener is a matter of calling `setOnClickPendingIntent()` on the `RemoteViews`:

```
remoteViews.setOnClickPendingIntent(R.id.refresh, refreshPI);
```

WRITING AN APP WIDGET

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
remoteViews.setOnClickListener(R.id.refresh, refreshPI)
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.kt](#))

`setOnClickListener()` takes the ID of the widget in the layout and the `PendingIntent`. If the user clicks that widget, Android will invoke the `PendingIntent` and will send our broadcast.

Reacting to the `PendingIntent`

Our broadcast is going to our `ColorWidget` as a broadcast. The entry point for a `BroadcastReceiver` is an `onReceive()` function. So, `ColorWidget` has its own `onReceive()` function to process our broadcast:

```
@Override
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(ACTION_REFRESH) &&
        intent.hasExtra(EXTRA_APP_WIDGET_ID)) {
        AppWidgetManager appWidgetManager =
            context.getSystemService(AppWidgetManager.class);

        updateWidget(
            context,
            appWidgetManager,
            intent.getIntExtra(EXTRA_APP_WIDGET_ID, -1)
        );
    }
    else {
        super.onReceive(context, intent);
    }
}
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
override fun onReceive(context: Context, intent: Intent) {
    if (intent.action == ACTION_REFRESH && intent.hasExtra(EXTRA_APP_WIDGET_ID)) {
        val appWidgetManager =
            context.getSystemService(AppWidgetManager::class.java)

        updateWidget(
            context,
            appWidgetManager,
            intent.getIntExtra(EXTRA_APP_WIDGET_ID, -1)
        )
    } else {

```

```
super.onReceive(context, intent)
}
}
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

`onReceive()` gets a copy of the `Intent` that was broadcast. The first thing that we do is to see if this is our refresh broadcast, by:

- Confirming that the action string is our custom refresh action, and
- Confirming that we have our app widget ID extra

If we do, we call `updateWidget()` from here, to update that app widget based on its ID. If we do not, we chain to the superclass implementation of `onReceive()`. In this case, the superclass is `AppWidgetProvider`, and it handles other broadcasts... such as the broadcast that eventually triggers the call to `onUpdate()`.

Displaying the BigSwatchFragment

The other action that we want to perform will be triggered by a click anywhere else on the app widget besides our refresh button. That should bring up our screen to show our color, the same as if the user tapped on a row in our list of colors.

In this case, we want to navigate to `BigSwatchFragment`. We are using Jetpack's Navigation framework for handling the navigation between fragments. One side benefit of that is we can use `NavDeepLinkBuilder` to create a `PendingIntent` for our use:

```
Bundle args = new Bundle();

args.putInt("color", color);

PendingIntent deepLinkPI = new NavDeepLinkBuilder(context)
    .setGraph(R.navigation.nav_graph)
    .setDestination(R.id.bigSwatchFragment)
    .setArguments(args)
    .createPendingIntent();

remoteViews.setOnClickPendingIntent(R.id.root, deepLinkPI);
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/samplerj/widget/ColorWidget.java](#))

```
val deepLinkPI = NavDeepLinkBuilder(context)
    .setGraph(R.navigation.nav_graph)
    .setDestination(R.id.bigSwatchFragment)
```


WRITING AN APP WIDGET

```
.setArguments(bundleOf("color" to color))  
.createPendingIntent()  
  
remoteViews.setOnClickPendingIntent(R.id.root, deepLinkPI)
```

(from [ColorWidget/src/main/java/com/commonsware/jetpack/sampler/widget/ColorWidget.kt](#))

NavDeepLinkBuilder takes:

- The ID of the navigation graph resource that we are using
- The ID of the destination that we are trying to navigate to
- A Bundle of arguments to pass to that destination (in this case, packaging up our color)

Then, a simple call to `createPendingIntent()` will create a `PendingIntent` that will start our activity and go to our desired destination. We can attach that `PendingIntent` to the `RemoteViews` via `setOnClickPendingIntent()`, and the rest is taken care of for us by the Navigation framework.

Part Three: Accessing Data

Thinking About Threads and LiveData

Users like apps that run smoothly. Users do not like applications that feel sluggish.

Usually, the reason why apps feel sluggish is due to problems with thread management, particularly when performing some form of I/O (disk, database, network, etc.). We need to take care when performing I/O that we do not slow down the UI of the app, while still getting the data that we need to display.

Nowadays, working with background threads is often done in conjunction with some form of reactive programming. In reactive programming, we request for work to be done and arrange to get control when it is completed. That work can be performed on one thread, while we get the results on some other thread. In particular, reactive programming attempts to hide a lot of that thread complexity, making it seem like we are doing “normal” programming.

This chapter introduces the key threading issues with Android and explores the Jetpack solution for reactive programming: LiveData. Mostly, this chapter is to “set the stage” for exploring I/O in upcoming chapters.

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.

That’s not really what happens.

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()`, `onCreateView()`, and similar functions are all called on the main application thread. While your code is executing in these functions, 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 in the app, the content does not scroll smoothly. Rather, it scrolls jerkily, interleaving periods of rapid movement with periods where the scrolling 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 widget-based UIs are updated at 60 frames per second. This means that any given frame needs to be assembled in less than 16ms. Since the framework and OS have work that needs to be done too, that means our application code must run in a lot less than 16ms per frame. And, since our app may be called on many times in a frame, this means that any given callback function, such as `onBindViewHolder()` of a `RecyclerView`, needs to be done *very* quickly, best measured in microseconds. If, instead, we take several milliseconds, we will “drop frames” (i.e., the screen does not get updated between two successive frames), and the user may perceive jank as a result.

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:

- Internet access, such as sending data to a Web service or downloading an image

- Significant file operations, since flash storage can be remarkably slow at times
- Any sort of complex calculations

The UI Thread is for UI

However, there is one big limitation: in general, you cannot modify the UI from a background thread. You can only modify the UI from the main application thread.

A few widgets have special support for being updated from a background thread. `ProgressBar` is one in particular that is set up this way. `ProgressBar` is designed to let you show progress of some background work, and so the Android engineers took the time to make `ProgressBar` support background UI updates.

However, outside of exceptions like that, any other UI updates have to be done from the main application thread.

This is a pain.

It means that we not only have to get our slow work off of the main application thread, but we have to update the UI with the results of that work on the main application thread. This sort of inter-thread communication is annoying.

Worse, this makes configuration changes that much trickier. Suppose we start some work on a background thread, and while that work is ongoing, the user rotates the screen. Our activity and its fragments get destroyed and recreated. Somehow, we not only need to arrange for our results to get processed on the main application thread, but they also need to be processed by whatever the now-current activity and fragments are.

This has been one of the major headaches that has plagued Android app developers since Android 1.0. Countless solutions have been offered by Google and by independent developers. The current “best practices” involve:

- Using a `ViewModel` (or similar construct) that is stable across configuration changes, so our code running on a background thread can hand the results to the `ViewModel` without having to worry as much about configuration changes
- Employing some sort of reactive programming solution for implementing that background work and getting the results over to the main application

thread

Introducing LiveData

However, even those “best practices” have a bit of a gap: how do we push data from a `ViewModel` to an activity or fragment? More importantly, how do we do so while taking into account configuration changes?

So far, our viewmodels have been simple state containers. The activities and fragments pull data from the viewmodels and push data into them.

Now, though, if we are saying that the viewmodel is somehow getting data delivered to it from the background, somehow we need the activity or fragment to find out that the data has arrived. So, for example, suppose that our fragment asks the viewmodel to retrieve some data from a Web site. That may take hundreds of milliseconds or longer, so the actual network I/O needs to be done on a background thread. When that I/O completes, the now-current fragment needs to find out about that, bearing in mind that it might be a different fragment instance by now, if a configuration change occurred while the I/O was ongoing.

The Jetpack solution to this problem is `LiveData`.

`LiveData` is itself a state container, designed to be held onto by a `ViewModel`. It has an Observer system, where interested parties can find out about changes in the data associated with the `LiveData`. So, if a background thread pushes new data into the `LiveData`, the `LiveData` will inform all of its observers. It does so on the main application thread, so observers do not need to worry about switching threads.

More importantly, `LiveData` is aware of lifecycles, such as those from activities and fragments. When an activity or fragment registers an Observer, and later the activity or fragment is destroyed, the `LiveData` automatically removes the associated Observer. Hence, when we undergo a configuration change, we do not need to remember to remove any Observer objects that we may have registered with `LiveData` — those get cleaned up automatically.

`LiveData` also automatically pushes an existing value, if there is one, to any new registered Observer objects. When coupled with the automatic-removal feature, this makes `LiveData` very convenient for dealing with configuration changes. The activities and fragments can largely forget about configuration changes entirely, just reacting to whatever the `LiveData` pushes to it. Those “pushes” will either be:

- Because some asynchronous work completed, or
- The device underwent a configuration change and the new activity or fragment is getting the data from before the change

Usually, the activity or fragment does not care about which of those scenarios occurred and can handle them identically.

Sources of LiveData

In many cases, you will get LiveData objects handed to you from something else. For example, if you use Room for working with on-device databases, you can have the results of your queries be in the form of LiveData. We will explore that more [later in the book](#).

Sometimes, you will use adapters to convert one form of reactive response into LiveData. For example, RxJava is the most popular reactive framework for Java. Jetpack offers a library that helps you convert RxJava reactive responses into LiveData, as RxJava itself knows nothing about Android and things like activity/fragment lifecycles. Similarly, with Kotlin coroutines, there are adapters supplied by the Jetpack, to help consume suspend functions and Flow via LiveData.

Otherwise, you are on your own for creating LiveData objects. For that, there are two main approaches:

1. Create a subclass of LiveData that handles the background threading and delivery of updates
2. Use MutableLiveData (which does not require a subclass the way LiveData does) and have code outside of that object deal with data sources, threading, etc.

Active and Inactive States

If a LiveData was instantiated in a forest, and nobody was there to observe data changes, does the LiveData really exist?

The answer is: yes, but it hopefully is not consuming any resources.

A LiveData implementation will be called with `onActive()` when it receives its first active observer. Here, “active” means that the activity or fragment registering the observer is in the started or resumed state. Conversely, the LiveData will be called with `onInactive()` once it no longer has any active observers, either because all

observers have been unregistered or none of them are active, as their lifecycles are all stopped or destroyed.

The idea is that a `LiveData` would only start consuming significant system resources — such as requesting GPS fixes — when there are active observers, releasing those resources when there are no more active observers. This works in many cases, though there are some that will require more finesse. For example, given that the GPS radio takes some time before it starts generating GPS fixes, a `LiveData` for GPS might want to wait some amount of time after losing its last active observer before releasing the GPS radio, in case a new observer pops up quickly, to avoid delays in getting those GPS fixes.

Subclasses of `LiveData`, therefore, should override `onActive()` (and perhaps `onInactive()`) and use those events to control resource consumption.

Colors... Live!

The `InLivingColor` sample module in the [Sampler](#) and [SamplerJ](#) projects implement the same basic UI as we saw in the `TwoActivities` sample [earlier in the book](#).

Part of the changes are to switch from two activities to two fragments and use the Navigation component. That largely mirrors what we saw [previously](#), so we will not focus on those changes here.

The change of importance for these samples are how we get our list of randomly-generated colors. In the earlier list-of-colors examples, we just generated colors directly in the `ColorViewModel`. This time, we are going to pretend that it takes “real work” to get these colors, such as having to call out to some random-color-generating Web service. To that end, we have a `ColorLiveData` subclass of `LiveData` that handles the threading aspects and generates the colors. `ColorViewModel` now exposes a `LiveData` of colors to observers, and a `ColorListFragment` pours those colors into the `RecyclerView` when they are ready.

ColorLiveData

Our `ColorLiveData` is responsible for creating a random set of colors, using a background thread. We only want to do this once we have one active observer — otherwise, the colors are pointless. So, our `LiveData` subclass overrides `onActive()` and handles the work there:

THINKING ABOUT THREADS AND LIVE DATA

```
package com.commonware.jetpack.samplerj.livedata;

import android.os.SystemClock;
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import androidx.lifecycle.LiveData;

class ColorLiveData extends LiveData<ArrayList<Integer>> {
    private final Random random = new Random();
    private final Executor executor = Executors.newSingleThreadExecutor();

    @Override
    public void onActive() {
        super.onActive();

        if (getValue() == null) {
            executor.execute(() -> {
                SystemClock.sleep(2000); // use only for book samples!
                postValue(buildItems());
            });
        }
    }

    private ArrayList<Integer> buildItems() {
        ArrayList<Integer> result = new ArrayList<>(25);

        for (int i = 0; i < 25; i++) {
            result.add(random.nextInt());
        }

        return result;
    }
}
```

(from [InLivingColor/src/main/java/com/commonware/jetpack/samplerj/livedata/ColorLiveData.java](#))

```
package com.commonware.jetpack.sampler.livedata

import android.os.SystemClock
import androidx.lifecycle.LiveData
import java.util.*
import java.util.concurrent.Executors

class ColorLiveData : LiveData<List<Int>>>() {
    private val random = Random()
    private val executor = Executors.newSingleThreadExecutor()
```

```
override fun onActive() {
    super.onActive()

    if (value == null) {
        executor.execute {
            SystemClock.sleep(2000) // use only for book samples!
            postValue(List(25) { random.nextInt() })
        }
    }
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/sampler/livedata/ColorLiveData.kt](#))

We use a Java Executor for performing our background work. For what we are doing now, a simple single-thread Executor is sufficient, but Executor gives us the flexibility to swap in a thread pool if we decided that we needed lots of random colors.

In `onActive()`, we see if we already have our colors. We do this by calling `getValue()`, which will return the value held by the `LiveData` or `null` if we have not yet generated any colors. If we do not already have colors, we use the Executor to run some code on a background thread to generate the colors.

We use `SystemClock.sleep()` to simulate two seconds worth of I/O to get these random colors, as our fictitious random-color-generating Web service is overloaded and our pretend Internet connection is poor. If you have used `Thread.sleep()` in Java before, `SystemClock.sleep()` works much the same way, blocking the current thread for the stated number of milliseconds. The biggest difference: `Thread.sleep()` throws an `InterruptedException`, which is unnecessary here and the resulting try/catch block just clutters up the example.

To update the `LiveData` with the new colors, we call `postValue()`. This updates the `LiveData` and — back on the main application thread — lets each of the registered observers know about the new data.

This particular `LiveData` only ever calls `postValue()` once. That is all this sample needs. However, there is nothing stopping you from having a `LiveData` that delivers a stream of updates, such as a stream of sensor readings. There, you might call `postValue()` hundreds or thousands of times, with the `LiveData` informing its observers each time.

ColorViewModel Changes

Our ColorViewModel now just holds onto a ColorLiveData instance:

```
package com.commonware.jetpack.samplerj.livedata;

import java.util.ArrayList;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.ViewModel;

public class ColorViewModel extends ViewModel {
    final LiveData<ArrayList<Integer>> numbers = new ColorLiveData();
}
```

(from [InLivingColor/src/main/java/com/commonware/jetpack/samplerj/livedata/ColorViewModel.java](#))

```
package com.commonware.jetpack.sampler.livedata

import androidx.lifecycle.ViewModel

class ColorViewModel : ViewModel() {
    val numbers = ColorLiveData()
}
```

(from [InLivingColor/src/main/java/com/commonware/jetpack/sampler/livedata/ColorViewModel.kt](#))

When the ColorViewModel is created, we create that ColorLiveData instance, and the viewmodel holds onto that instance for the lifetime of the viewmodel. That way, even after the work is completed and we have our list of colors, we still hold onto those colors across configuration changes. However, to keep this example simple, we are skipping the saved-state logic to try to handle cases where the process is terminated shortly after the user moves the app to the background.

Since we do not start our background work until `onActive()` is called on the ColorLiveData, just creating the ColorLiveData is cheap. And, if for some reason we never observe the ColorLiveData, we do not go through the “expense” of generating this list of random colors.

Observing the Colors

In the earlier list-of-colors example, we would call `submitList()` on our ColorAdapter for our colors with the numbers that we get from the ColorViewModel. Now, we *observe* the `numbers` property, as that is a LiveData.

In Java, we use the `observe()` method on `LiveData` itself:

```
package com.commonware.jetpack.samplerj.livedata;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.commonware.jetpack.samplerj.livedata.databinding.FragmentListBinding;
import androidx.annotation.NonNull;
import androidx.fragment.app.Fragment;
import androidx.lifecycle.ViewModelProvider;
import androidx.navigation.fragment.NavHostFragment;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

public class ColorListFragment extends Fragment {
    private FragmentListBinding binding;

    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
        binding = FragmentListBinding.inflate(inflater, container, false);

        return binding.getRoot();
    }

    @Override
    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        ColorViewModel vm = new ViewModelProvider(this).get(ColorViewModel.class);

        ColorAdapter colorAdapter = new ColorAdapter(getLayoutInflater(),
            this::navTo);

        binding.items.setLayoutManager(new LinearLayoutManager(requireContext()));
        binding.items.addItemDecoration(new DividerItemDecoration(requireContext(),
            DividerItemDecoration.VERTICAL));
        binding.items.setAdapter(colorAdapter);

        vm.numbers.observe(getViewLifecycleOwner(), colorAdapter::submitList);
    }

    @Override
    public void onDestroyView() {
        super.onDestroyView();

        binding = null;
    }

    private void navTo(int color) {
        NavHostFragment.findNavController(this)
            .navigate(ColorListFragmentDirections.showColor(color));
    }
}
```

(from [InLivingColor/src/main/java/com/commonware/jetpack/samplerj/livedata/ColorListFragment.java](#))

observe() on a LiveData takes two parameters:

1. A LifecycleOwner, usually supplied by the activity or fragment that is using the LiveData, so the LiveData knows when it is active or inactive
2. An Observer implementation

An Observer has a single method, observe(), that returns void and takes an instance of whatever type the LiveData holds. So, in our case, observe() will be passed an ArrayList<Int>. Here, in the Java scenario, we use a Java 8 method reference to hand that ArrayList over to the ColorAdapter, whereas in Kotlin, we just use an ordinary lambda expression:

```
package com.commonware.jetpack.sampler.livedata

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.viewModels
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonware.jetpack.sampler.livedata.databinding.FragmentListBinding
import com.commonware.jetpack.sampler.util.ViewBindingFragment

class ColorListFragment :
    ViewBindingFragment<FragmentListBinding>(FragmentListBinding::inflate) {
    private val vm: ColorViewModel by viewModels()

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        val colorAdapter = ColorAdapter(layoutInflater) { color ->
            navTo(color)
        }

        useBinding { binding ->
            binding.items.apply {
                layoutManager = LinearLayoutManager(activity)

                addItemDecoration(
                    DividerItemDecoration(
                        activity,
                        DividerItemDecoration.VERTICAL
                    )
                )

                adapter = colorAdapter
            }
        }
    }
}
```

```
    }  
  }  
  
  vm.numbers.observe(viewLifecycleOwner) { colorAdapter.submitList(it) }  
}  
  
private fun navTo(color: Int) {  
    findNavController().navigate(ColorListFragmentDirections.showColor(color))  
}  
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/sampler/livedata/ColorListFragment.kt](#))

We will look more at the `LifecycleOwner` parameter, and what `viewLifecycleOwner` is, [shortly](#).

The Results

When the fragment is first created, it creates the `ColorViewModel`, which creates a `ColorLiveData`. When the fragment then calls `observe()` on the `LiveData`, and the fragment is later shown on the screen, the `ColorLiveData` will be called with `onActive()`, as the fragment will now be started. At that point, the `ColorLiveData` will generate the colors (after a two-second delay). When `ColorLiveData` calls `postValue()`, the `Observer` registered by the fragment will be called, and the `ColorAdapter` will get its colors.

If the user rotates the screen, our original `ColorListFragment` will be destroyed and recreated. When the fresh `ColorListFragment` gets its `ColorViewModel`, it will be the already-existing `ColorLiveData` instance. When the fragment calls `observe()`, its `Observer` will be called immediately if the `ColorLiveData` already has its colors — that is handled automatically by `LiveData`. Our fragment's code does not care whether it is the first or second instance of the `ColorListFragment` — it gets the colors the same way and consumes them the same way.

Sources of Owners

`observe()` on a `LiveData` takes a `LifecycleOwner`. Technically, you could implement the `LifecycleOwner` interface on just about anything. In practice, there are three commonly-used `LifecycleOwner` implementations:

1. `FragmentActivity` and things that inherit from it, like `AppCompatActivity`
2. `Fragment`

3. The “view LifecycleOwner” of a Fragment

The first two are fairly straightforward. They let the LiveData know about the lifecycle of the activity and the fragment, respectively.

However, as we noted [earlier](#) fragments are weird, with their pair of lifecycles:

1. The lifecycle of the fragment itself
2. The lifecycle of a particular UI managed by that fragment

The general rule of thumb is:

- If you are observing a LiveData solely for populating a UI, such as we are doing here in `ColorListFragment`, use `getViewLifecycleOwner()` as the LifecycleOwner and start observing in `onViewCreated()` (or possibly in `onCreateView()`)
- If you are observing a LiveData and need to observe for the entire lifetime of the fragment, regardless of its UI, use the Fragment itself as the LifecycleOwner and start observing in `onCreate()`... but do not attempt to update any views in response to that LiveData, as they may not exist at the time you receive an updated value

The first pattern is by far the most common one.

Where Do Threads Come From? Um, Besides From Me?

In this sample, we created our own thread, by means of `Executors.newSingleThreadExecutor()`. This will be needed in some situations, where you are using a synchronous API, one that offers no sort of callback or other asynchronous option.

However, frequently, something else might create the threads for you.

Threads from Reactive Frameworks

As mentioned earlier, RxJava is the most popular reactive framework for Java development. With RxJava, you indicate what work you want done and what RxJava-supplied thread pool that you want that work to be done on (e.g., “use the thread pool dedicated for I/O”). RxJava takes care of allocating the threads and using them

as appropriate.

In Kotlin, coroutines are gaining in popularity. Coroutines, along with other Kotlin language and standard library features, offer a lot of the same capabilities as does RxJava. Kotlin coroutines, though, can support Kotlin's array of platform targets (e.g., JavaScript) in addition to Android, while RxJava is tied to Java-based apps. On the other hand, coroutines are fairly new to Kotlin overall, whereas RxJava has been around longer. But, if you use coroutines, once again you will not need to create threads yourself — you describe where you want work to be done, and the coroutines engine handles the actual threading.

Threads from Data Sources

Many libraries that support database and network I/O offer their own thread pools for performing that I/O. They offer either a true reactive API or a classic callback-based system for getting the I/O results asynchronously.

Room, mentioned earlier in this chapter, is the Jetpack solution for on-device database access. Room wraps Android's SQLite support in an API that, among other things, supports both synchronous and reactive access. If you prefer, you can call synchronous APIs and handle the threading yourself. Or, you can have Room respond with `LiveData`, interact with RxJava, or support Kotlin coroutines. We will explore Room more [later in the book](#).

There are a massive number of third-party libraries available for Internet access. Some are general-purpose for accessing data over HTTPS, such as `OkHttp`. Others are optimized for specific types of Web access:

- Images (e.g., `Glide`, `Picasso`)
- REST Web services (`Retrofit`)
- GraphQL Web services (`Apollo-Android`)

All offer asynchronous APIs. In some cases, they offer integration with reactive frameworks like RxJava. In others, they just allow you to supply a callback function to be invoked when the I/O is completed.

Threads from Background Processing

Some work needs to be done in the background as a result of the user doing something in your UI. For example, a Twitter client needs to refresh the timeline when the user clicks a refresh button or performs a pull-to-refresh gesture. In those

cases, using threads and LiveData, RxJava, or Kotlin coroutines are recommended.

In other cases, the work may need to happen somewhat later, even if the user is no longer interacting with the app. For example, you may have a business requirement to check a server once per hour, every hour. Here, the user is not triggering the need for background work — the passage of time is.

WorkManager is the Jetpack solution for this problem. It handles threading for you; your work will be performed on a background thread automatically. We will explore WorkManager more [later in the book](#).

Coroutines and ViewModel

Most of the Kotlin samples in this book that need to do background processing will do so using coroutines. We will have suspend functions that either call other suspend functions or do work in a designated thread or thread pool:

```
suspend fun doSomethingCool(): AwesomeMix = withContext(Dispatchers.IO) {  
    // TODO something that returns an AwesomeMix  
}
```

A suspend function needs to be called inside of a CoroutineScope. While Kotlin offers a GlobalScope that can be used for anything, ideally you use a scope that is more closely tied to the work that is being done.

In Android UI development, that frequently will be a CoroutineScope associated with a ViewModel. There is a viewModelScope extension property available to ViewModel that provides a CoroutineScope tied to the lifetime of the ViewModel. In particular, if the ViewModel is no longer being used (i.e., it is being called with `onCleared()`), the `viewModelScope` is also cleaned up.

So, you wind up with code like this:

```
fun doSomethingCoolAndGetTheResultsOnTheMainApplicationThread() {  
    viewModelScope.launch(Dispatchers.Main) {  
        val mix = repo.doSomethingCool()  
  
        // TODO something with mix on the main application thread  
    }  
}
```

(where `repo` is an object of the class that has the aforementioned

`doSomethingCool()` function)

`Dispatchers.Main` will mean that while `doSomethingCool()` will run on a background thread, we will get the result delivered to us on the main application thread.



You can learn more about coroutines in the "Introducing Coroutines" chapter of [Elements of Kotlin Coroutines](#)!

Some of the Kotlin examples later in the book will demonstrate using coroutines.

The End of LiveData?!?

LiveData has been around for a few years as the official Google reactive API solution.

Nowadays, LiveData is still the official Google reactive API solution... for Java.

For Kotlin developers, Google is starting to steer you towards `StateFlow`. `StateFlow` is very similar to `LiveData`, but it is more tightly tied into the world of coroutines. It offers greater power and flexibility, but it is only practical for use in Kotlin projects.

We will see `StateFlow` — and a similar construct, `SharedFlow` — [later in the book](#).

Adding Some Architecture

MVC. MVP. MVVM. MVI. These abbreviations get tossed around a lot in app development discussions, and increasingly in Android app development discussions. Those using these abbreviations often think that:

- Everybody knows what they mean, and
- There is a single universal definition for each of those abbreviations, one that everybody holds

In reality, these MV* abbreviations are well-known in some circles and unknown in others. And, even among people who think they know these abbreviations, there is a fair bit of disagreement about what the abbreviations mean, particularly when it comes time to writing actual code.

This book is not a book on GUI architectures. However, in this chapter, we will explore some facets of Google's architecture recommendations.

Repositories

One key to Google's recommended architecture pattern is the repository. Rather than the GUI code dealing with things like disk and network I/O, we delegate that to a repository, which handles those details. The GUI code just works with the repository, one typically implemented as a singleton object (a global instance that all pieces of the app can use).

Objective: Isolation

We really want our GUI code to be separate from our I/O code.

Mostly, that is for testing purposes. When we want to test our GUI code, we may or may not want to actually go through that I/O:

- I/O is slow, and being able to use some sort of mock repository that is purely memory-backed would allow tests to run faster
- Server I/O can be a problem, as there may not be a server that you can use for automated testing
- We often need to test specific scenarios, and it is much easier to do that if we can have a mock repository that we can control, rather than have to set up files (or, worse, server entries) to get the real repository to respond the way our scenario requires

Repository Structures

Simple repositories — like the ones that will be shown in this book — work directly with files, databases, servers, and the like.

Some projects will add another layer of indirection, where repositories work with “data sources”. A data source would be responsible for the direct I/O with one particular set of files, or a particular database, or a particular Web service. The repository would interact with the data source.

Mostly, the data source abstraction is there for complex repositories:

- You want to add in-memory caching at the repository level, and therefore you want to separate out the I/O logic into another class
- You need to support both local I/O (for a persistent cache) and network I/O (the source of shared or updated data), and putting all of that in a single repository results in a huge class
- The actual data source itself is user-configurable, both in terms of details (e.g., email account information) and protocol (e.g., IMAP4, JMAP, POP3), where the repository does not know at compile-time what data source implementation will be needed

So, while the apps in this book largely will skip the repository/data source separation of concerns, that is just because the sample apps are pretty tiny.

Unidirectional Data Flow

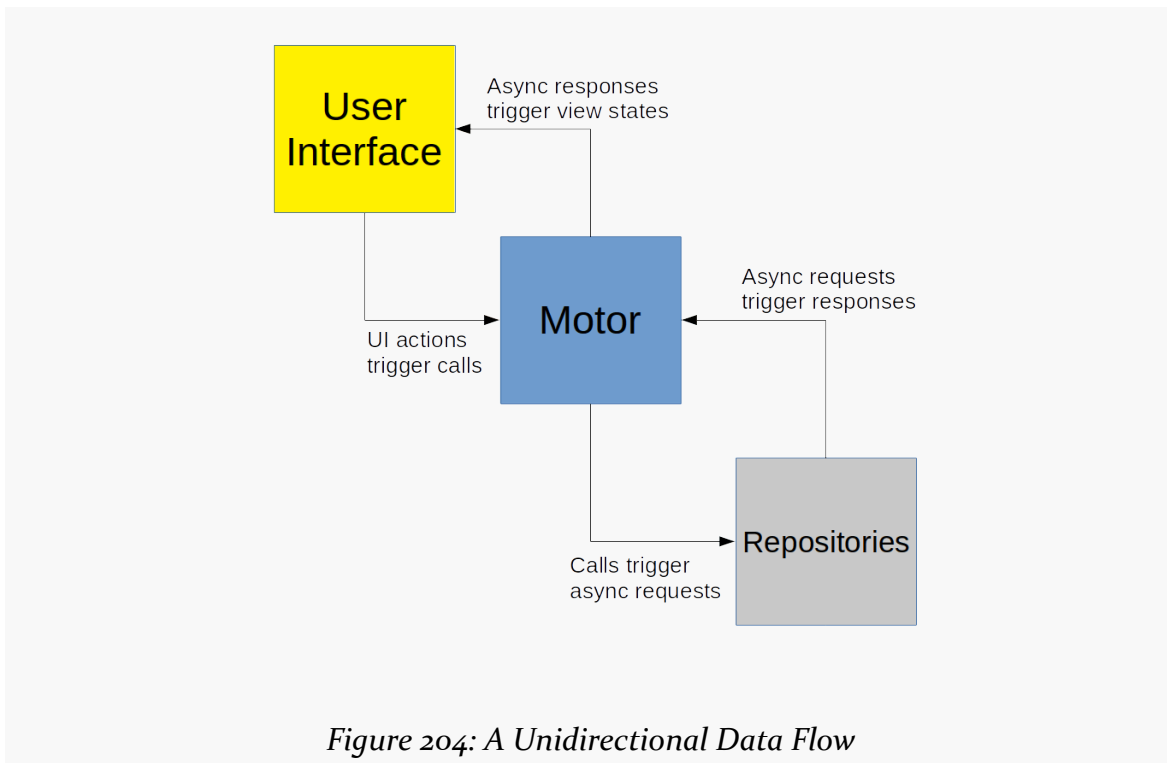
Many of the remaining examples in this book employ a [unidirectional data flow](#)

ADDING SOME ARCHITECTURE

pattern. [Popularized by Redux](#) in the world of Web app development, adopting a unidirectional data flow can help simplify how different bits of your app work with that data.

While the details will differ somewhat by implementation, the general approach of a unidirectional data flow (UDF) is to minimize the number of places in the app that can *change* data. In particular, while a user interface may allow the user to change data, the GUI code itself does not change data, but instead works off of a stream of immutable objects as a source of details for what to render in the GUI itself.

In particular, the UDF approach that will be used in most of the remaining book samples will look a bit like this:



In a nutshell:

- When a user does something in the UI, such as click a toolbar button, the UI calls a function on something called “the motor”
- In response to those calls, the motor asks the repository to read, update, or delete some data, using some sort of asynchronous mechanism
- The result of the repository work gets delivered to the motor via that

asynchronous mechanism

- The motor emits a “view state” of details that get used by the UI to update what the user sees

A UDF Implementation

That UDF explanation may make more sense once we work through an example.

The DiceLight sample module in the [Sampler](#) and [SamplerJ](#) projects implement a “diceware” app. It allows the user to randomly generate a passphrase from a word list. Later, we will look at [a more complex Diceware sample](#) that allows the user to supply an alternative word list. For now, we will settle for having a word list packaged with the app.

The UI

When you launch the app, you are immediately greeted by a random passphrase:

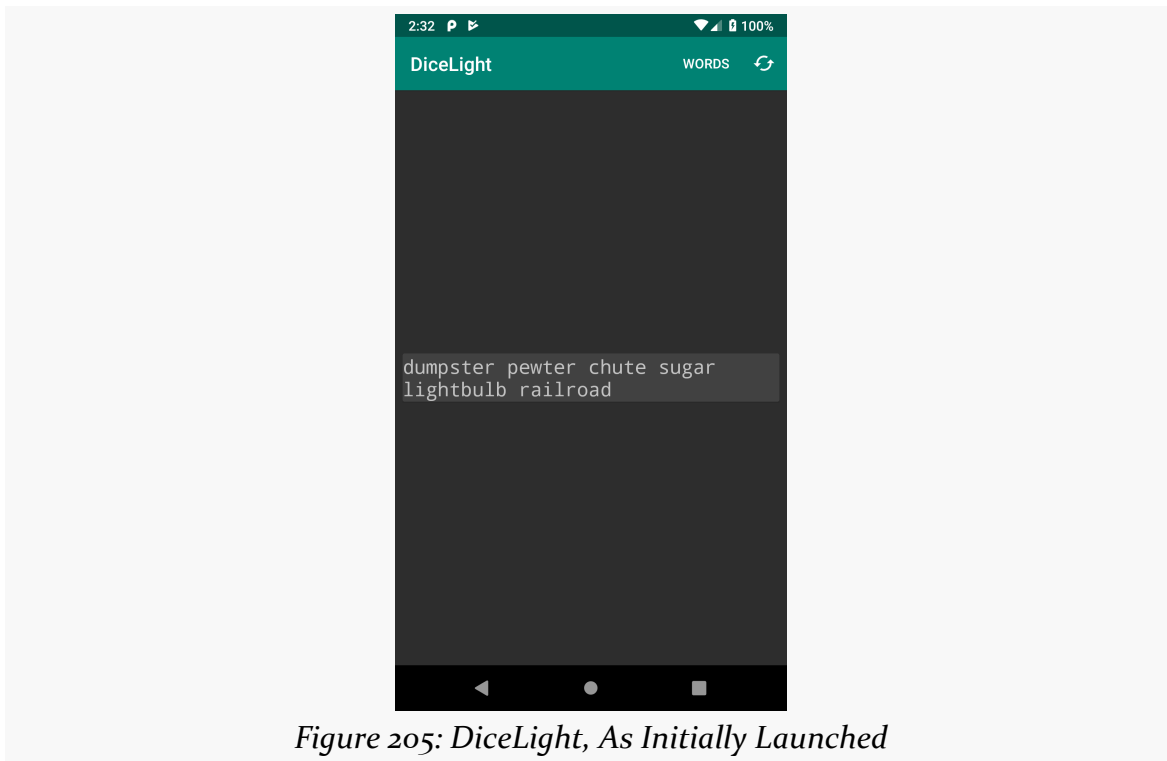


Figure 205: DiceLight, As Initially Launched

The refresh toolbar button will generate a fresh passphrase, in case you do not like

the original one. The “WORDS” item will display a list of word counts:

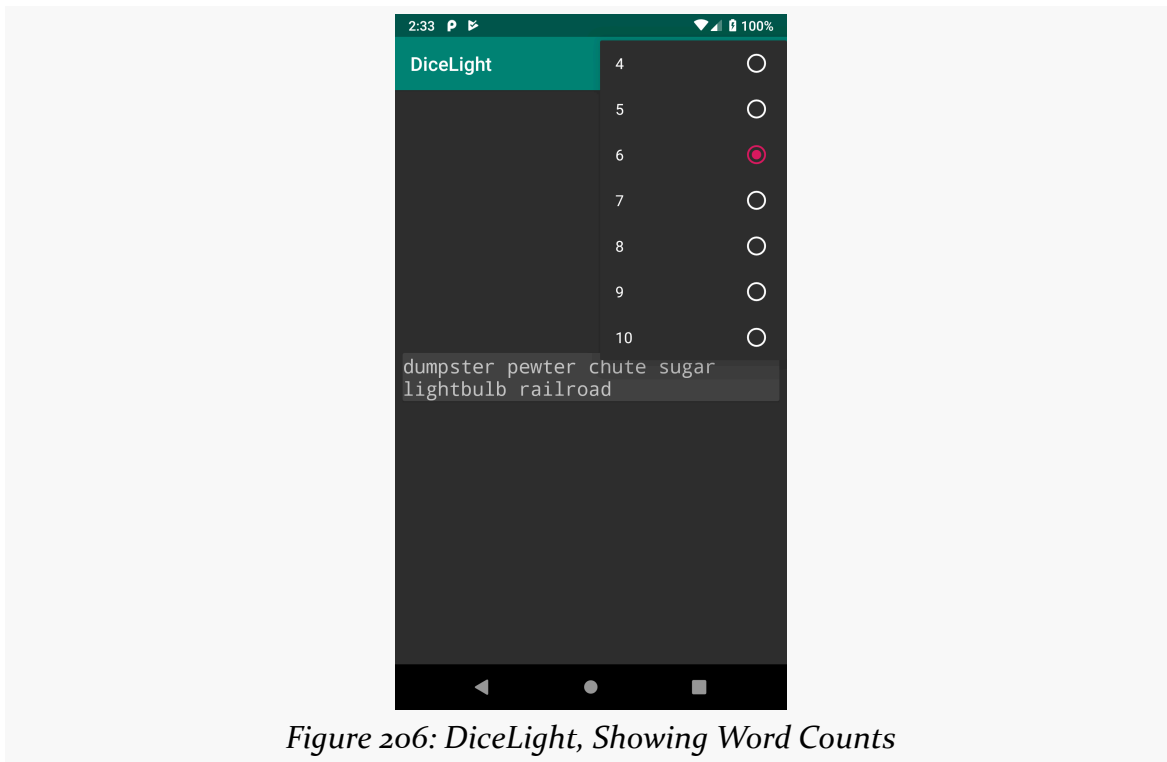


Figure 206: DiceLight, Showing Word Counts

Switching to a different word count will give you a fresh passphrase with that number of words, in case you want one that is shorter or longer.

The words come from [the EFF's short wordlist](#), where we randomly choose a few out of the 1,296 in the list. That word list is packaged in the `assets/` directory, so it is part of the Android APK that is our compiled app.

The View-State

In order to render this UI, we need a randomly-generated passphrase. However, there are two other scenarios to consider:

- What happens while we are loading the words? Initially, they are not in memory, and until they are, we cannot randomly generate a passphrase from them. That disk I/O will be quick, but not instantaneous. The speed of network I/O is a lot worse, though in this case we do not have any of that. So, in theory, we should show something, such as a `ProgressBar`, while we are loading the words.

ADDING SOME ARCHITECTURE

- What happens if we have some sort of problem loading the words? That should not happen here, as we know that our words are available in our APK. However, in general, disk I/O and network I/O can trigger exceptions. If we have such an exception, we really ought to tell the user about it.

So, now we have three pieces of data to track:

- Whether or not we are loading
- The current passphrase, if we have it
- The current error, if we have one

Our `MainViewState` encapsulates those three pieces of data... but how it does so varies by language.

Java

In Java, `MainViewState` is a simple class:

```
package com.commonware.jetpack.diceware;

class MainViewState {
    final boolean isLoading;
    final String content;
    final int wordCount;
    final Throwable error;

    MainViewState(boolean isLoading, String content, int wordCount, Throwable error) {
        this.isLoading = isLoading;
        this.content = content;
        this.wordCount = wordCount;
        this.error = error;
    }
}
```

(from [DiceLight/src/main/java/com/commonware/jetpack/diceware/MainViewState.java](https://github.com/Commonware/dice-light/blob/master/src/main/java/com/commonware/jetpack/diceware/MainViewState.java))

We will use `null` to indicate that we do not have our passphrase (content) yet or do not have a current error.

Kotlin

That Java representation works, and it is simple, but it is not great.

Partly, the problem is that those three pieces of data are mutually exclusive to an extent:

ADDING SOME ARCHITECTURE

- If we have our passphrase, we did not crash, so we will not have an error
- If we have an error, we will not have a passphrase
- If we are loading, we will not have either of those things

Plus, using `null` as a “we don’t have the data yet” value is annoying in Kotlin. You keep having to use safe calls (`?.`) and the like to deal with `null`.

In Kotlin, we take a substantially different approach, using a sealed class:

```
package com.commonware.jetpack.diceware

sealed class MainViewState {
    object Loading : MainViewState()
    data class Content(val passphrase: String, val wordCount: Int) : MainViewState()
    data class Error(val throwable: Throwable) : MainViewState()
}
```

(from [DiceLight/src/main/java/com/commonware/jetpack/diceware/MainViewState.kt](#))

Now, any instance of `MainViewState` has just the data that is appropriate for it (e.g., the passphrase and word count for `Content`). And, for a state that has no data — such as `Loading` — it is an object, rather than a class.

This loading/content/error pattern is increasingly common in Android, as a way of representing the three major states.

The Repository

So, we need to create those states and get them to the activity, so the activity can do something with that information to display in the UI. A lot of that work is going to be handled by our repository.

The `PassphraseRepository` is responsible for loading the words and generating a random subset for us on demand. The core logic is the same between the Java and Kotlin implementations, but the API is different, owing to different ways in handling background work and singletons.

Java

Our `PassphraseRepository` is set up as a classic Java singleton, using a static volatile field and a synchronized, lazy-initializing `get()` method:

ADDING SOME ARCHITECTURE

```
package com.commonware.jetpack.diceware;

import android.content.Context;
import android.content.res.AssetManager;
import com.google.common.util.concurrent.ListenableFuture;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicReference;
import androidx.concurrent.futures.CallbackToFutureAdapter;

class PassphraseRepository {
    private static final String ASSET_FILENAME = "eff_short_wordlist_2_0.txt";
    private static volatile PassphraseRepository INSTANCE;

    synchronized static PassphraseRepository get(Context context) {
        if (INSTANCE == null) {
            INSTANCE = new PassphraseRepository(context.getApplicationContext());
        }

        return INSTANCE;
    }

    private final AssetManager assets;
    private final AtomicReference<List<String>> wordsCache =
        new AtomicReference<>();
    private final SecureRandom random = new SecureRandom();
    private final Executor threadPool = Executors.newSingleThreadExecutor();

    private PassphraseRepository(Context context) {
        assets = context.getAssets();
    }

    ListenableFuture<List<String>> generate(int wordCount) {
        return CallbackToFutureAdapter.getFuture(completer -> {
            threadPool.execute(() -> {
                try {
                    List<String> words = wordsCache.get();

                    if (words == null) {
                        InputStream in = assets.open(PassphraseRepository.ASSET_FILENAME);
```

```
        words = readWords(in);
        in.close();

        synchronized (wordsCache) {
            wordsCache.set(words);
        }
    }

    completer.set(rollDemBones(words, wordCount));
}
catch (Throwable t) {
    completer.setException(t);
}
});

return "generate words";
});
}

private List<String> rollDemBones(List<String> words, int wordCount) {
    List<String> result = new ArrayList<>();
    int size = words.size();

    for (int i = 0; i < wordCount; i++) {
        result.add(words.get(random.nextInt(size)));
    }

    return result;
}

private 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;
}
}
```

ADDING SOME ARCHITECTURE

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

`get()` takes a `Context` as a parameter. We need a `Context` to be able to read in our word list stored in assets. However, we do not want to hold onto an arbitrary `Context` in this singleton — if the `Context` is an `Activity`, we will wind up with a memory leak. So, we hold onto the `Application` singleton edition of `Context` instead.

The singleton has a single visible method: `generate()`. Given a word count, `generate()` generates a list of randomly-chosen words. However, this may involve disk I/O to read in the assets, if this is the first time we are trying to generate a passphrase in this process. As a result, we use a background thread for that work, in the form of an `Executor` created from `Executors.newSingleThreadExecutor()`. The `execute()` method that we call takes a `Runnable` (here implemented as a Java 8 lambda expression) and does that work on that background thread.

However, we need to be able to get the list of words to the caller when that background task completes. To that end, `generate()` returns a `ListenableFuture`. A `Future` is an object that represents some outstanding work; `ListenableFuture` is one that lets one register a listener to find out when that work is done. `CallbackToFutureAdapter` is a Jetpack utility class that helps you create a `ListenableFuture`. Specifically, in the `Resolver` that you pass to `getFuture()` (here implemented as a Java 8 lambda expression), you are passed a “completer” where you can:

- Provide the data to be returned by the background work (`set()`), or
- Provide a `Throwable` if something went wrong (`setException()`)

The lambda expression needs to return a `String`, but this is only used for logging purposes.

`ListenableFuture` and `CallbacktoFutureAdapter` are obtained via the `androidx.concurrent:concurrent-futures` library:

```
}
```

(from [DiceLight/build.gradle](#))

The repository holds onto the read-in words in a cache, wrapped by an `AtomicReference` to ensure that we handle possible parallel access to the cache across multiple threads. `generate()` will:

- See if we have cached the words

ADDING SOME ARCHITECTURE

- If not, read in the words and save them in the cache
- Randomly choose the requested number of words, passing that List to the caller via the ListenableFuture

The word list contains a “die roll” value and a word for each line, separated by tabs, such as:

```
1154 again
1155 agency
1156 aggressor
1161 aghast
1162 agitate
1163 agnostic
```

The `readWords()` method needs to split each line along the whitespace and take the second part.

Kotlin

The Kotlin implementation does the same work, with three key differences:

1. Kotlin offers an `object` keyword for creating singletons, so we use that rather than manage our own singleton
2. Since such singletons cannot have a constructor, we pass the `Context` into `generate()`, whereas Java supplied it to the repository constructor
3. Rather than use `ListenableFuture`, `generate()` is a suspend function, doing the work on a background thread, and we can still return our result list or throw an exception

```
package com.commonware.jetpack.diceware

import android.content.Context
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import java.io.BufferedReader
import java.io.InputStream
import java.io.InputStreamReader
import java.security.SecureRandom
import java.util.concurrent.atomic.AtomicReference

private const val ASSET_FILENAME = "eff_short_wordlist_2_0.txt"

object PassphraseRepository {
    private val wordsCache = AtomicReference<List<String>>()
```

ADDING SOME ARCHITECTURE

```
private val random = SecureRandom()

suspend fun generate(context: Context, count: Int): List<String> {
    val words: List<String>? = wordsCache.get()

    return words?.let { rollDemBones(it, count) }
        ?: loadAndGenerate(context, count)
}

private suspend fun loadAndGenerate(
    context: Context,
    count: Int
): List<String> =
    withContext(Dispatchers.IO) {
        val inputStream = context.assets.open(ASSET_FILENAME)

        inputStream.use {
            val words = it.readLines()
                .map { line -> line.split("\t") }
                .filter { pieces -> pieces.size == 2 }
                .map { pieces -> pieces[1] }

            wordsCache.set(words)

            rollDemBones(words, count)
        }
    }

private fun rollDemBones(words: List<String>, wordCount: Int) =
    List(wordCount) { words[random.nextInt(words.size)] }

private fun InputStream.readLines(): List<String> {
    val result = mutableListOf<String>()

    BufferedReader(InputStreamReader(this)).forEachLine { result.add(it); }

    return result
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](https://github.com/commonsware/jetpack-diceware/blob/master/PassphraseRepository.kt))

As a result, our repository is a bit simpler (no singleton code, plus lots of Kotlin functions to make it easier to read in and process the words). And, since we are handling the background thread in the repository, consumers of this repository can be a bit simpler as well.

The Motor

In Android, we have a problem with our UIs: they keep getting destroyed, courtesy of configuration changes. We saw how we can use a `ViewModel` for retaining objects across configuration changes, and we saw how we can have a `ViewModel` expose `LiveData` objects to the UI layer to deliver data updates.

A motor is simply a `ViewModel` exposing `LiveData`.

The reason for the “motor” name (e.g., `MainMotor`) instead of a “viewmodel” name (e.g., `ViewModel`) comes from other GUI architectures. In MVVM (Model-View-Viewmodel) and MVP (Model-View-Presenter), what gets referred to as the “viewmodel” fills the sort of role that we have here as the view-state: it is the data to be displayed in the UI. However, in Android’s Architecture Components, just because we use a `ViewModel` (for configuration changes) does not mean we want that object to serve as a viewmodel (representing the data to be displayed). So, this book uses “motor” to identify a `ViewModel` that exposes a `LiveData` of view-state objects.

And, as with our repository, the motor is a bit different between Java and Kotlin, principally due to the way threading is handled.

Java

In Java, our repository gives us a `ListenableFuture`. Since that API involves I/O, we need to get its work onto a background thread. So, `MainMotor` adapts the `ListenableFuture` to `LiveData`:

```
package com.commonware.jetpack.diceware;

import android.app.Application;
import android.text.TextUtils;
import com.google.common.util.concurrent.ListenableFuture;
import java.util.List;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;

public class MainMotor extends AndroidViewModel {
    private static final int DEFAULT_WORD_COUNT = 6;
    private final MutableLiveData<MainViewState> viewStates =
        new MutableLiveData<>();
    private final PassphraseRepository repo;
```


ADDING SOME ARCHITECTURE

```
public MainMotor(@NonNull Application application) {
    super(application);

    repo = PassphraseRepository.get(application);
    generatePassphrase(DEFAULT_WORD_COUNT);
}

LiveData<MainViewState> getViewStates() {
    return viewStates;
}

void generatePassphrase() {
    final MainViewState current = viewStates.getValue();

    if (current == null) {
        generatePassphrase(DEFAULT_WORD_COUNT);
    }
    else {
        generatePassphrase(current.wordCount);
    }
}

void generatePassphrase(int wordCount) {
    viewStates.setValue(new MainViewState(true, null, wordCount, null));

    ListenableFuture<List<String>> future = repo.generate(wordCount);

    future.addListener(() -> {
        try {
            viewStates.postValue(new MainViewState(false,
                TextUtils.join(" ", future.get()), wordCount, null));
        }
        catch (Exception e) {
            viewStates.postValue(new MainViewState(false, null, wordCount, e));
        }
    }, Runnable::run);
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainMotor.java](https://github.com/commonsware/jetpack-diceware/blob/master/MainMotor.java))

MainMotor has a MutableLiveData for our MainViewState, exposing it to the UI layer via a getViewStates() method returning a LiveData. MainMotor also has reference to the PassphraseRepository singleton. And, since we need to supply a Context to the repository, MainMotor extends AndroidViewModel, so we have a getApplication() method to retrieve the Application singleton Context.

The `generatePassphrase(int)` method first emits a view-state indicating that we are loading. Then, it gets the `ListenableFuture` from the repository via a call to `generate()`. Next, it adds a listener to the `ListenableFuture`, to be notified when the work is complete. This takes a `Runnable` to be invoked at that time, plus an `Executor` implementation to control what thread is used to execute the `Runnable`. In our case, courtesy of Java 8 method references, we can replace that `Executor` with a `Runnable::run` method reference, to say “execute the `Runnable` on whatever thread you are on, please”.

Then, in the `Runnable` (lambda expression), we:

- Use Android’s `TextUtils.join()` method to convert the list of words into a single space-delimited passphrase
- Emit a view-state with that result, or emit a view-state with the exception if we ran into a problem

We use `postValue()` on `MutableLiveData`, to be sure that no matter what thread the `Runnable` executes, it is safe for us to update the `MutableLiveData`.

We also have a `generatePassphrase()` method that takes no parameters. This will use the `wordCount` from the previous view-state. If there was no previous view-state, it uses an overall default value.

Note that `MainMotor` immediately generates a passphrase, using the default word count, once it is created. This way, the activity does not need to tell the motor to do anything — the activity just observes the `LiveData` and reacts when the initial passphrase is emitted.

Kotlin

In Kotlin, since our repository exposes a suspend function, we use `viewModelScope` to launch our coroutine, receiving the results on the main application thread:

```
package com.commonware.jetpack.diceware

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

private const val DEFAULT_WORD_COUNT = 6
```

ADDING SOME ARCHITECTURE

```
class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _results = MutableLiveData<MainViewState>()
    val results: LiveData<MainViewState> = _results

    init {
        generatePassphrase(DEFAULT_WORD_COUNT)
    }

    fun generatePassphrase() {
        generatePassphrase(
            (results.value as? MainViewState.Content)?.wordCount ?: DEFAULT_WORD_COUNT
        )
    }

    fun generatePassphrase(wordCount: Int) {
        _results.value = MainViewState.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = try {
                val randomWords = PassphraseRepository.generate(
                    getApplication(),
                    wordCount
                )

                MainViewState.Content(randomWords.joinToString(" "), wordCount)
            } catch (t: Throwable) {
                MainViewState.Error(t)
            }
        }
    }
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainMotor.kt](#))

We also:

- Use the PassphraseRepository singleton supplied by the object declaration
- Use Kotlin's own `joinToString()` instead of Android's `TextUtils.join()` to convert the list of words into the passphrase
- Use our sealed class implementations, so we are emitting some sub-type of `MainViewState`

Otherwise, this works the same as its Java equivalent, including the two varieties of `generatePassphrase()` (one with an explicit word count, one without) and generating a passphrase when the `MainMotor` is created.

The Activity

Our layout is based on a `CardView`: a widget that is a simple rounded rectangle with a bit of a drop shadow. Inside of there, we have a `TextView` and a `ProgressBar`:

ADDING SOME ARCHITECTURE

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp">

    <androidx.cardview.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" />

        <ProgressBar
            android:id="@+id/progress"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </androidx.cardview.widget.CardView>

</FrameLayout>
```

(from [DiceLight/src/main/res/layout/activity_main.xml](#))

Managing that layout is a simple activity. We could have used a fragment here, but there is only one screen. In general, the remaining sample apps will use fragments and the Navigation component when there is more than one screen, but just a single Activity otherwise.

In `onCreate()`, we get our `MainMotor`, start observing the `LiveData`, apply the `MainViewState` to our widgets:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ActivityMainBinding binding =
        ActivityMainBinding.inflate(getLayoutInflater());

    setContentView(binding.getRoot());

    motor = new ViewModelProvider(this).get(MainMotor.class);
```

ADDING SOME ARCHITECTURE

```
motor.getViewStates().observe(this, viewState -> {
    binding.progress.setVisibility(
        viewState.isLoading ? View.VISIBLE : View.GONE);

    if (viewState.content != null) {
        binding.passphrase.setText(viewState.content);
    }
    else if (viewState.error != null) {
        binding.passphrase.setText(viewState.error.getLocalizedMessage());
        Log.e("Diceware", "Exception generating passphrase",
            viewState.error);
    }
    else {
        binding.passphrase.setText("");
    }
});
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityMainBinding.inflate(layoutInflater)

    setContentView(binding.root)

    motor.results.observe(this) { viewState ->
        when (viewState) {
            MainViewState.Loading -> {
                binding.progress.visibility = View.VISIBLE
                binding.passphrase.text = ""
            }
            is MainViewState.Content -> {
                binding.progress.visibility = View.GONE
                binding.passphrase.text = viewState.passphrase
            }
            is MainViewState.Error -> {
                binding.progress.visibility = View.GONE
                binding.passphrase.text = viewState.throwable.localizedMessage
                Log.e(
                    "Diceware",
                    "Exception generating passphrase",
                    viewState.throwable
                )
            }
        }
    }
}
```

ADDING SOME ARCHITECTURE

```
    }  
  }  
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](https://github.com/commonsware/jetpack-diceware/blob/master/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt))

We also have a menu resource for the word count options and the refresh button:

```
<?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/word_count"  
    app: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"  
    app:showAsAction="ifRoom"
```

ADDING SOME ARCHITECTURE

```
        android:title="@string/menu_refresh" />
    </menu>
```

(from [DiceLight/src/main/res/menu/actions.xml](#))

The word_count menu item is a bit unusual, following the Android recipe for creating such a selection menu:

- The <item> has a <menu> child...
- ...which in turn holds a <group> with android:checkableBehavior="single"...
- ... which wraps a set of <item> elements, one for each checkable menu item...
- ... with android:checked="true" on the word_count_6 item, to pre-check that one

We then have code in onCreateOptionsMenu() and onOptionsItemSelected() on MainActivity to set up and handle that menu:

```
@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.refresh:
            motor.generatePassphrase();
            return true;

        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());
            motor.generatePassphrase(Integer.parseInt(item.getTitle().toString()));

            return true;
    }
}
```

ADDING SOME ARCHITECTURE

```
return super.onOptionsItemSelected(item);  
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {  
    menuInflater.inflate(R.menu.actions, menu)  
  
    return super.onCreateOptionsMenu(menu)  
}  
  
override fun onOptionsItemSelected(item: MenuItem): Boolean {  
    when (item.itemId) {  
        R.id.refresh -> {  
            motor.generatePassphrase()  
            return true  
        }  
  
        R.id.word_count_4, R.id.word_count_5, R.id.word_count_6, R.id.word_count_7,  
        R.id.word_count_8, R.id.word_count_9, R.id.word_count_10 -> {  
            item.isChecked = !item.isChecked  
  
            motor.generatePassphrase(Integer.parseInt(item.title.toString()))  
  
            return true  
        }  
    }  
  
    return super.onOptionsItemSelected(item)  
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

The refresh item is simple: we just call `generatePassphrase()` again, causing the motor to get a fresh set of words from the repository and emitting another `MainViewState` to update our UI.

For the word count, we have more work to do. First, we need to toggle the `isChecked` state of the `MenuItem`, because Android (inexplicably) does not handle that for us when the user clicks a checkable menu item. Then, we cheat a bit and parse the actual text of the menu item as an `Integer` — this only works because this app is English-only, so we know that the menu item captions can be parsed by `Integer.parseInt()`. We then call `generatePassphrase()` to generate a passphrase with the new number of words.

Revisiting the Unidirectional Data Flow

Our activity takes user input (activity launch, refresh, word-count change) and converts those into function calls on the motor:

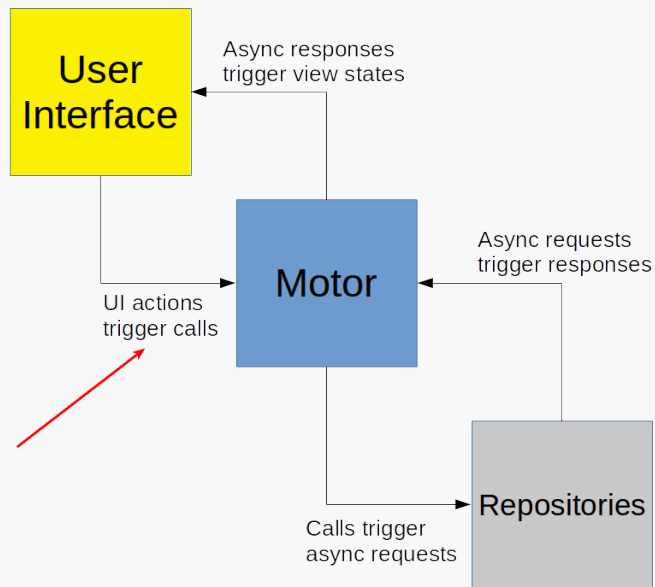
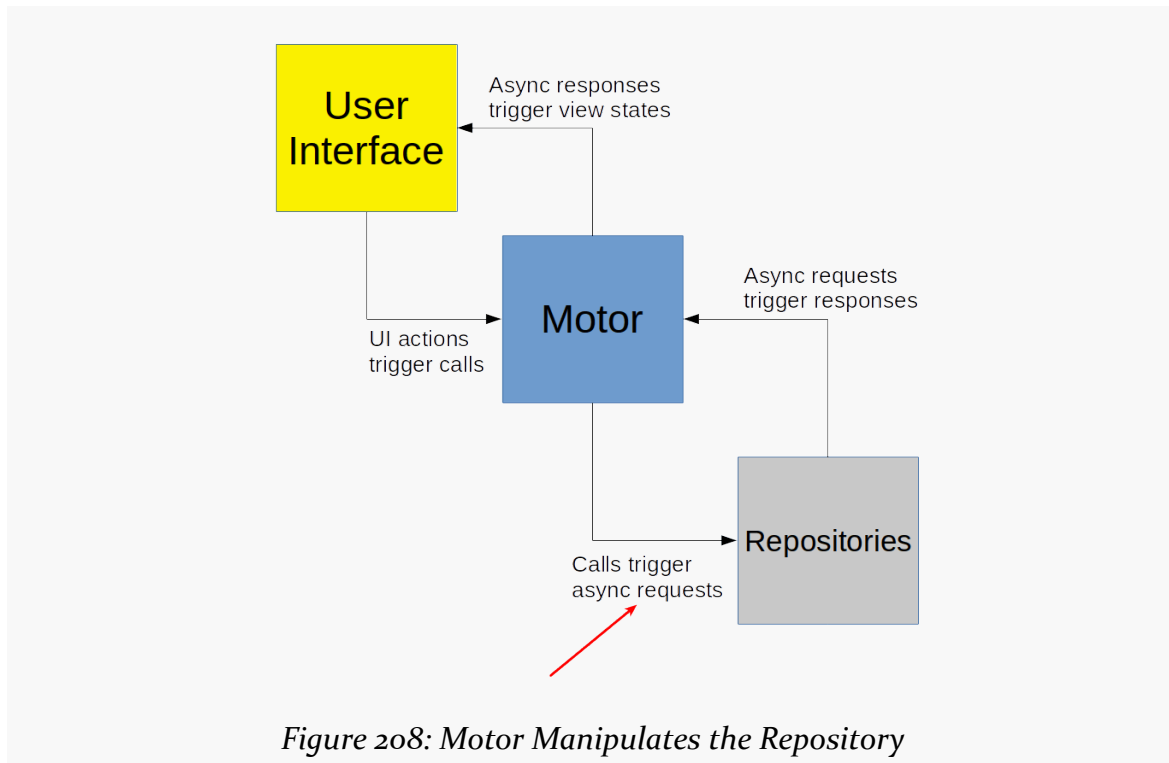


Figure 207: UI Events Trigger Calls on Motor

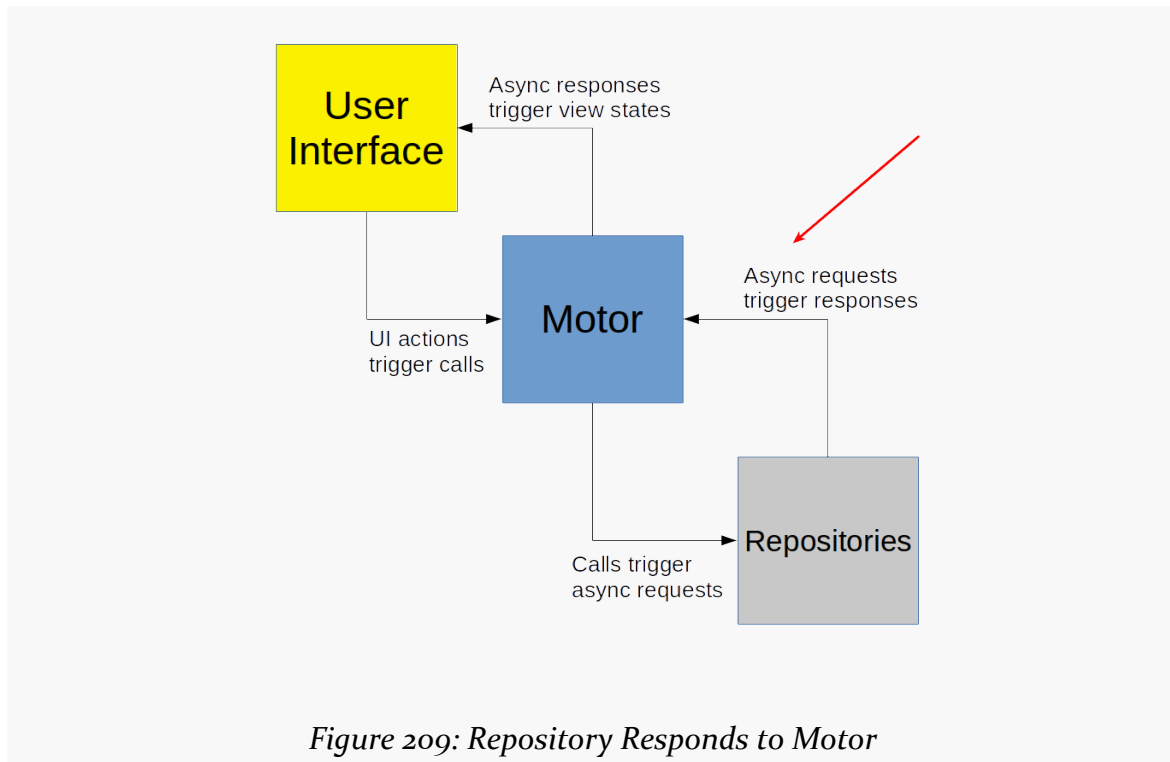
ADDING SOME ARCHITECTURE

The motor then converts those calls into operations to be performed on our repository, in an asynchronous fashion:



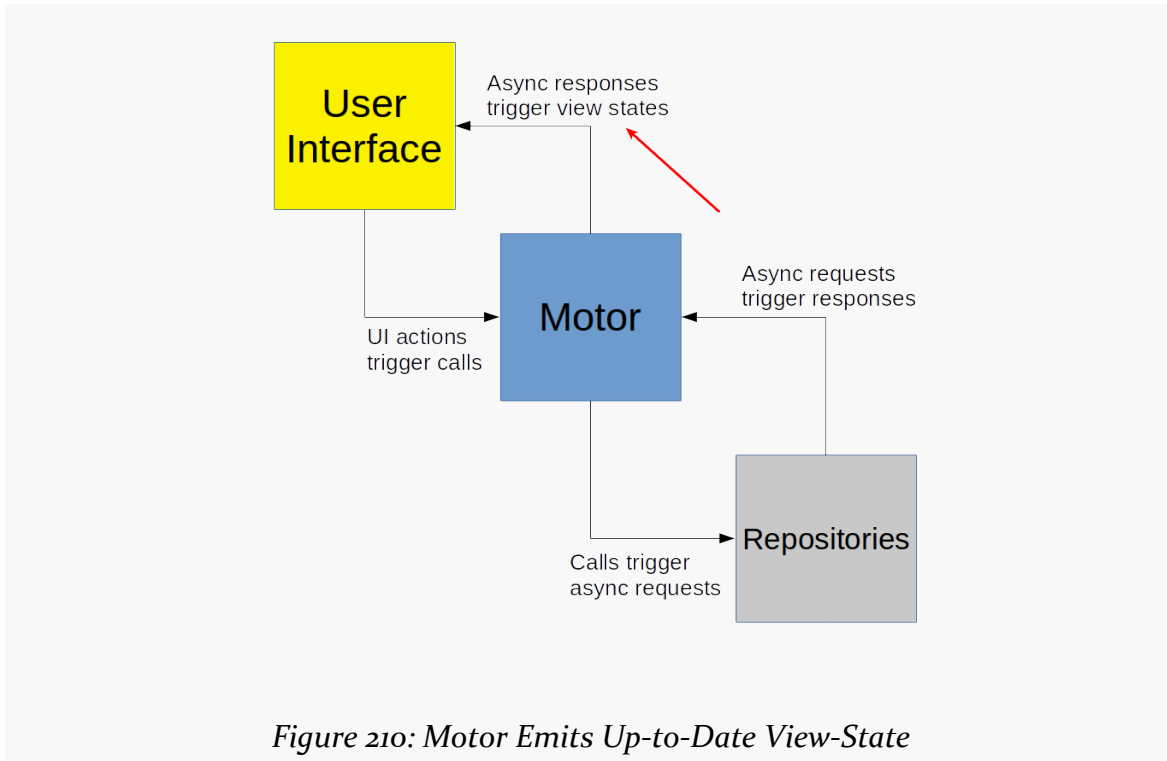
ADDING SOME ARCHITECTURE

The repository does the work and asynchronously delivers the result to the motor:



ADDING SOME ARCHITECTURE

The motor takes those results and crafts a revised view-state that the activity can use to render the UI:



Obviously, this is a very simple example. There are plenty of MVI frameworks, such as Spotify's [Mobius](#), that provide a scaffold around this sort of flow that you can plug into. However, in many cases, that level of sophistication (and corresponding complexity) is not needed, and a simple motor-based flow like the one shown here can suffice.

The Kotlin LiveData Alternative: StateFlow

As was noted back in [the chapter on threads](#), Google is starting to push StateFlow as an alternative to LiveData for use in Kotlin projects.



You can learn more about StateFlow and SharedFlow in the "Opting Into SharedFlow and StateFlow" chapter of [Elements of Kotlin Coroutines](#)!

Fundamentally, `StateFlow` works just like `LiveData`:

- `StateFlow` caches a last-seen value
- When the value changes, in addition to caching it, `StateFlow` passes the value to all current observers
- A new observer immediately gets the last-seen value, in addition to any changes while the observer is observing

The `DiceFlow` sample module in the [Sampler](#) project is a clone of `DiceLight`, except that it uses `StateFlow` in places where `DiceLight` uses `LiveData`. Hence, the repository and the view-state are unaffected, since they do not use `LiveData`. What are affected are `MainMotor` and `MainActivity`.

Impacts on `MainMotor`

Just as we have `MutableLiveData` and `LiveData`, we have `MutableStateFlow` and `StateFlow`. `MainMotor` in `DiceFlow` is identical to its equivalent in `DiceLight`, just with `_results` and `results` based off of `MutableStateFlow` and `StateFlow`:

```
package com.commonware.jetpack.diceware

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.ViewModelScope
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.launch

private const val DEFAULT_WORD_COUNT = 6

class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _results = MutableStateFlow<MainViewState>(MainViewState.Loading)
    val results = _results.asStateFlow()

    init {
        generatePassphrase(DEFAULT_WORD_COUNT)
    }

    fun generatePassphrase() {
        generatePassphrase(
            (results.value as? MainViewState.Content)?.wordCount ?: DEFAULT_WORD_COUNT
        )
    }

    fun generatePassphrase(wordCount: Int) {
        _results.value = MainViewState.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = try {
```

ADDING SOME ARCHITECTURE

```
val randomWords = PassphraseRepository.generate(
    getApplication(),
    wordCount
)

MainViewState.Content(randomWords.joinToString(" "), wordCount)
} catch (t: Throwable) {
    MainViewState.Error(t)
}
}
```

(from [DiceFlow/src/main/java/com/commonsware/jetpack/diceware/MainMotor.kt](#))

As with `MutableLiveData`, we use `setValue()` to update the value in the `MutableStateFlow`. `setValue()` is a suspend function, so it needs to be called inside of a coroutine, but we already had a coroutine for use in calling `PassphraseRepository.generate()`, so that did not introduce a code change.

`MutableLiveData` supports being instantiated without a starting value. `MutableStateFlow` does not, so we initialize it to the Loading view-state.

And, to get a `StateFlow` from the `MutableStateFlow`, we call `asStateFlow()`.

Impacts on MainActivity

`MainActivity` still observes results and passes the view-states to a `when()` to route them to appropriate rendering logic. However, consuming a `StateFlow` is a bit different than consuming a `LiveData`:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    lifecycleScope.launch {
        lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
            motor.results.collect { viewState ->
                when (viewState) {
                    MainViewState.Loading -> {
                        binding.progress.visibility = View.VISIBLE
                        binding.passphrase.text = ""
                    }
                    is MainViewState.Content -> {
                        binding.progress.visibility = View.GONE
                    }
                }
            }
        }
    }
}
```

```
        binding.passphrase.text = viewState.passphrase
    }
    is MainViewState.Error -> {
        binding.progress.visibility = View.GONE
        binding.passphrase.text = viewState.throwable.localizedMessage
        Log.e(
            "Diceware",
            "Exception generating passphrase",
            viewState.throwable
        )
    }
}
}
```

(from [DiceFlow/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](https://github.com/commonsware/jetpack-diceware/blob/master/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt))

We call `collect()` on the `StateFlow`, passing in our lambda expression to be invoked for each view-state, including the initial Loading one.

That `collect()` gets wrapped in `repeatOnLifecycle()`, called on the `Lifecycle` object representing this fragment's view lifecycle. We will only collect events while we are in the started or resumed states `repeatOnLifecycle(Lifecycle.State.STARTED)`, so we will not try processing view-states while our UI is in the background. And that in turn is wrapped in `launch()` on the `LifecycleScope` associated with this fragment's view lifecycle, so the entire coroutine can be canceled cleanly when the fragment's views are destroyed.

States and Events

Back in [the chapter on dialogs](#), we had a scenario that sounds similar to the view-states that we have seen in this chapter. We had a `ConfirmationDialogFragment` displaying an `AlertDialog`, and we wanted to get the button-click events from the `ConfirmationDialogFragment` back to the `MainFragment` that triggered the dialog to be shown.

It may seem like we can do the same thing there as we did here: use `LiveData` and have the `ConfirmationDialogFragment` emit a view-state that the `MainFragment` observes. In truth, that is close to what we need but not quite the same, because “states” and “events” are subtly different, particularly in Android.

Definitions

In `MainFragment`, if the user clicks the positive button in the dialog, we want to show a Toast. A Toast has its own lifecycle: it appears and vanishes on its own.

What we do *not* want to do is display a Toast again after a configuration change. The Toast should appear exactly once per positive button click. If the user clicks the positive button, sees the Toast, then rotates the screen, we should not re-display the Toast.

As a result, the “did the user click the positive button” output from `ConfirmationDialogFragment` cannot readily be part of a view-state. A view-state represents data that we *want* to survive a configuration change. If `MainActivity` wanted to update its button caption based on whether the user had accepted the dialog in its most recent invocation, then we would want the “did the user click the positive button” to be part of the view-state, so we could show the right caption after a configuration change. That does not work well in our Toast scenario: `MainFragment` would get the most-recent view-state after a configuration change, see that the positive button had been clicked, and show the Toast again.

In many respects, the problem itself is not the configuration change. In theory, what could happen is:

- The user clicks the really big button to display the dialog
- The user clicks the positive button on that dialog, then immediately rotates the screen, so fast that `MainFragment` has not had time to react to the positive button click

In that case, we still do want `MainFragment` to show the Toast... but only once.

In the terminology used in this book, we have “states” and “events”:

- A state is something that we want to survive a configuration change and use
- An event is something that we want to use exactly once, regardless of any configuration changes that may or may not occur

Impacts on Delivery

`LiveData` is a value holder with a way to tell a set of observers when that value changes. It is designed for states, which is why we used it for the view-state in this chapter.

On its own, though, it is not well-suited for events. There is no built-in concept of “consuming” an event from LiveData that would prevent that event from being consumed again after a configuration change.

The Jetpack does not have a great solution for this problem. The current “state of the art” for handling this varies based on your programming language.

Java: Single Live Event

Google’s original solution was what is called the “single live event” pattern. And, while Google has been backtracking away from that solution, it is still the best option for Java projects that have not adopted RxJava as a reactive API, where RxJava projects would use PublishSubject or similar options.

The single live event pattern still has us deliver via LiveData. However, we wrap the data that represents the event (e.g., a boolean of whether the user dismissed the dialog via the positive button or not) in a wrapper that tracks whether or not we have consumed the event.

In the Java edition of the [NukeFromOrbit](#) sample module, that wrapper is called Event:

```
package com.commonware.jetpack.samplerj.dialog;

import androidx.lifecycle.Observer;

public final class Event<T> {
    public interface Handler<T> {
        void handle(T content);
    }

    public static class EventObserver<T> implements Observer<Event<T>> {
        private final Event.Handler<T> handler;

        public EventObserver(Handler<T> handler) {
            this.handler = handler;
        }

        @Override
        public void onChanged(Event<T> event) {
            if (event != null) {
                event.handle(handler);
            }
        }
    }
}
```

ADDING SOME ARCHITECTURE

```
}

private boolean hasBeenHandled = false;
private final T content;

public Event(T content) {
    this.content = content;
}

private void handle(Event.Handler<T> handler) {
    if (!hasBeenHandled) {
        hasBeenHandled = true;
        handler.handle(content);
    }
}
}
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/samplerj/dialog/Event.java](#))

At its core, `Event` wraps some object (using generic type `T`), referred to as `content`. It has a `handle()` method that takes a callback (`Event.Handler`) and calls that callback if the `Event` has not already been handled. `Event` also provides an `Observer` implementation called `EventObserver` that handles an event received from a `LiveData`, if that event has not already been handled.

When the user clicks one of the dialog buttons, or uses the system BACK button to dismiss the dialog, `ConfirmationDialogFragment` calls either `onAccept()` or `onDecline()` on the `GraphViewModel`:

```
package com.commonsware.jetpack.samplerj.dialog;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class GraphViewModel extends ViewModel {
    private MutableLiveData<Event<Boolean>> results = new MutableLiveData<>();

    LiveData<Event<Boolean>> getResultStream() {
        return results;
    }

    void onAccept() {
        results.postValue(new Event<>(true));
    }
}
```

ADDING SOME ARCHITECTURE

```
void onDecline() {
    results.postValue(new Event<>(false));
}
}
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/samplerj/dialog/GraphViewModel.java](#))

GraphViewModel in turn posts a boolean value to a MutableLiveData called results, where the boolean is wrapped in an Event.

MainFragment then sets up an EventObserver to receive those boolean values:

```
vm.getResultStream().observe(getViewLifecycleOwner(),
    new Event.EventObserver<>(wasAccepted -> {
        if (wasAccepted) {
            Toast.makeText(requireContext(), "BOOOOOOOM!",
                Toast.LENGTH_LONG).show();
        }
    }));
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/samplerj/dialog/MainFragment.java](#))

So, when the user clicks a button, the EventObserver handles the Event and passes the underlying value to our code. If the user then rotates the screen or otherwise triggers a configuration change, while EventObserver itself will receive the Event from the LiveData, since the Event will have been marked as having been handled, EventObserver does not wind up calling our code again. So, we get the boolean value exactly once per button click.

This works. It is a bit of a hack, but it works.

Kotlin: SharedFlow

The “bit of a hack” aspect is why Google would prefer that you use something else. In Kotlin, that “something else” right now is SharedFlow and MutableSharedFlow.

GraphViewModel in Kotlin still has onAccept() and onDecline() functions that ConfirmationDialogFragment calls. This time, though, they offer() a boolean value to a MutableSharedFlow named results:

```
package com.commonsware.jetpack.sampler.dialog

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
```

ADDING SOME ARCHITECTURE

```
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.asSharedFlow
import kotlinx.coroutines.launch

class GraphViewModel : ViewModel() {
    private val results = MutableSharedFlow<Boolean>()
    val resultStream = results.asSharedFlow()

    fun onAccept() {
        viewModelScope.launch { results.emit(true) }
    }

    fun onDecline() {
        viewModelScope.launch { results.emit(false) }
    }
}
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/sampler/dialog/GraphViewModel.kt](#))

A SharedFlow works a bit like LiveData and StateFlow, in that it can have one or more observers and passes any offered object to each of them. Unlike LiveData and StateFlow, a SharedFlow does not hold onto the last-offered object, so observers get each object exactly once. The relationship between SharedFlow and MutableSharedFlow is very much like the relationship between LiveData and MutableLiveData: MutableSharedFlow has a read/write API, whereas SharedFlow has a read-only API.

The MutableSharedFlow is part of the GraphViewModel implementation. Its API is in the form of a SharedFlow created from that MutableSharedFlow. The So, MainFragment consumes those click results via that SharedFlow:

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
        vm.resultStream.collect { wasAccepted ->
            if (wasAccepted) {
                Toast.makeText(requireContext(), "BOOOOOOOM!", Toast.LENGTH_LONG)
                    .show()
            }
        }
    }
}
```

(from [NukeFromOrbit/src/main/java/com/commonsware/jetpack/sampler/dialog/MainFragment.kt](#))

This works much like the EventObserver from the Java example. The lambda expression that we pass to collect() will get called for each boolean offered by

ADDING SOME ARCHITECTURE

ConfirmationDialogFragment and GraphViewModel. And, as with the StateFlow example earlier in this chapter:

- `collect()` gets wrapped in `repeatOnLifecycle()`, called on the `Lifecycle` object representing this fragment's view lifecycle
- That in turn is wrapped in `launch()` on the `LifecycleScope` associated with this fragment's view lifecycle, so the entire coroutine can be canceled cleanly when the fragment's views are destroyed

Working with Content

If you have written software for other platforms, you are used to working with files on the filesystem. Android has some support for that, but it is limited and awkward — we will explore that more [in an upcoming chapter](#).

Instead, Google would like for us to work with content more generally, whether that content comes from files, from services like Google Drive, or other places. To that end, Android comes with the Storage Access Framework for creating and consuming content.

The Storage Access Framework

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 or to include in an MMS message.

Somehow, that email or text messaging client needs to allow the user to choose a photo. Some developers attempt to do this by looking for photo files on the filesystem, but that will miss lots of photos, particularly on newer versions of Android. Some developers will use a class called `MediaStore` to query for available photos. That is a reasonable choice, but `MediaStore` only allows you to query for

certain types of content, such as photos. If the user wants to attach a PDF to the email, `MediaStore` is not a great solution. Also, `MediaStore` only knows about local content, not items in cloud services used by the user.

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. Document providers simply publish details about their available content — including items 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 document 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).

More Dice!

The Diceware modules of the [Java](#) and [Kotlin](#) projects are based on the `DiceLight` sample that we saw [previously](#). The difference is that in this version of the app, there is an overflow menu with “Open Word File”, where the user can choose a different word list to use as the source of words for the passphrases. To let the user choose the word list, we will use the Storage Access Framework.

The SAF Actions

From a programming standpoint, the Storage Access Framework feels like the “file open”, “file save-as”, and “choose directory” dialogs that you may be used to from other GUI environments. The biggest difference is that the Storage Access Framework is not limited to files.

These three bits of UI are tied to three Intent actions:

Action	Equivalent Role
<code>ACTION_OPEN_DOCUMENT</code>	file open
<code>ACTION_CREATE_DOCUMENT</code>	file save-as
<code>ACTION_OPEN_DOCUMENT_TREE</code>	choose directory

Those three Intent actions are designed for use with `startActivityForResult()`. In `onActivityResult()`, if we got a result, that will contain a `Uri` that points to a

document (for `ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT`) or document tree (for `ACTION_CREATE_DOCUMENT`). We can then use that `Uri` to write content, read in existing content, or find child documents in a tree.

Currently, Google recommends using `registerForActivityResult()` with a suitable `ActivityResultContracts` and `ActivityResultCallback`, instead of using `startActivityForResult()` directly. Fortunately, the Jetpack provides three contracts, one for each action:

Contract	Action
<code>ActivityResultContracts.OpenDocument</code>	<code>ACTION_OPEN_DOCUMENT</code>
<code>ActivityResultContracts.CreateDocument</code>	<code>ACTION_CREATE_DOCUMENT</code>
<code>ActivityResultContracts.OpenDocumentTree</code>	<code>ACTION_OPEN_DOCUMENT_TREE</code>

Opening a Document

Technically, we do not “open” a document using `ActivityResultContracts.OpenDocument`. Instead, we are requesting a `Uri` pointing to some document that the user chooses. So, our Diceware samples set up an `openDoc` object to represent that contract and request:

```
private final ActivityResultLauncher<String[]> openDoc =
    registerForActivityResult(new ActivityResultContracts.OpenDocument(),
        new ActivityResultCallback<Uri>() {
            @Override
            public void onActivityResult(Uri uri) {
                motor.generatePassphrase(uri);
            }
        });
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
private val openDoc = registerForActivityResult(ActivityResultContracts.OpenDocument()) {
    motor.generatePassphrase(it)
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

Then, when the user taps on an “Open” app bar item, we call `launch()` on `openDoc` to actually trigger the request to open a document:

WORKING WITH CONTENT

```
openDoc.launch(new String[]{"text/plain"});
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
openDoc.launch(arrayOf("text/plain"))
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

This time, though, the contract requires some input: an array of strings, where each string represents either a concrete MIME type (e.g. `text/plain`) or a wildcard MIME type (e.g., `text/*`). In our case, we pass in a single `text/plain` MIME type, limiting users to choosing that sort of file.

The user is presented with the system's `ACTION_OPEN_DOCUMENT` UI to browse among various places and choose a document:

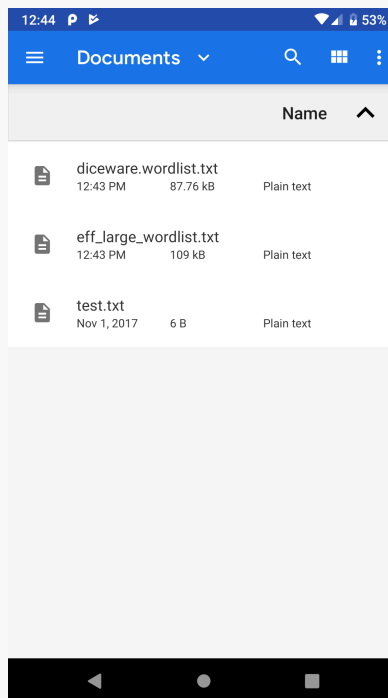


Figure 211: Storage Access Framework UI, Showing Documents

Our callbacks will receive a `Uri` pointing to the chosen document. We pass that to `MainMotor`, asking it to generate a passphrase for us, using that `Uri` and the user's current requested word count.

If you want to try this yourself, you can download the [diceware.wordlist.txt](#) or [eff_large_wordlist.txt](#) word list files, put them on your test device, then open

them up within the Diceware app.

Consuming the Chosen Content

MainMotor is largely the same as before. However, when we generate the passphrase, we pass the Uri along to generate() on PassphraseRepository:

```
package com.commonware.jetpack.diceware;

import android.app.Application;
import android.net.Uri;
import android.text.TextUtils;
import com.google.common.util.concurrent.ListenableFuture;
import java.util.List;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.MutableLiveData;

public class MainMotor extends AndroidViewModel {
    private static final int DEFAULT_WORD_COUNT = 6;
    private final PassphraseRepository repo;
    private Uri wordsDoc = PassphraseRepository.ASSET_URI;
    final MutableLiveData<MainViewState> viewStates =
        new MutableLiveData<>();

    public MainMotor(@NonNull Application application) {
        super(application);

        repo = PassphraseRepository.get(application);
        generatePassphrase(DEFAULT_WORD_COUNT);
    }

    void generatePassphrase() {
        final MainViewState current = viewStates.getValue();

        if (current == null) {
            generatePassphrase(DEFAULT_WORD_COUNT);
        }
        else {
            generatePassphrase(current.wordCount);
        }
    }

    void generatePassphrase(int wordCount) {
        viewStates.setValue(new MainViewState(true, null, wordCount, null));
    }
}
```

WORKING WITH CONTENT

```
    ListenableFuture<List<String>> future = repo.generate(wordsDoc, wordCount);

    future.addListener((Runnable)() -> {
        try {
            viewStates.postValue(new MainViewState(false,
                TextUtils.join(" ", future.get()), wordCount, null));
        }
        catch (Exception e) {
            viewStates.postValue(new MainViewState(false, null, wordCount, e));
        }
    }, Runnable::run);
}

void generatePassphrase(Uri wordsDoc) {
    this.wordsDoc = wordsDoc;

    generatePassphrase();
}
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainMotor.java](#))

```
package com.commonsware.jetpack.diceware

import android.app.Application
import android.net.Uri
import androidx.lifecycle.*
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

private const val DEFAULT_WORD_COUNT = 6

class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _results = MutableLiveData<MainViewState>()
    val results: LiveData<MainViewState> = _results
    private var wordsDoc = ASSET_URI

    init {
        generatePassphrase(DEFAULT_WORD_COUNT)
    }

    fun generatePassphrase() {
        generatePassphrase(
            (results.value as? MainViewState.Content)?.wordCount ?: DEFAULT_WORD_COUNT
        )
    }

    fun generatePassphrase(wordCount: Int) {
        _results.value = MainViewState.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = try {
                val randomWords = PassphraseRepository.generate(
                    getApplication(),
                    wordsDoc,

```

WORKING WITH CONTENT

```
        wordCount
    )

    MainViewState.Content(randomWords.joinToString(" "), wordCount)
} catch (t: Throwable) {
    MainViewState.Error(t)
}
}
}

fun generatePassphrase(wordsDoc: Uri) {
    this.wordsDoc = wordsDoc

    generatePassphrase()
}
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainMotor.kt](#))

For our initial passphrase, though, we start off with a `Uri` that points to our asset:

```
static final Uri ASSET_URI =
    Uri.parse("file:///android_asset/eff_short_wordlist_2_0.txt");
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

```
val ASSET_URI: Uri =
    Uri.parse("file:///android_asset/eff_short_wordlist_2_0.txt")
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

In `PassphraseRepository`, our words cache now is an `LruCache`. This is an Android SDK class, representing a thread-safe `Map` that caps its size to a certain number of entries. If we try putting more things in the cache than we have room for, the `LruCache` will evict the least-recently-used (“LRU”) entry. In our case, the cache is keyed by a `Uri` and is capped to at most four word lists:

```
private final LruCache<Uri, List<String>> wordsCache = new LruCache<>(4);
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

```
private val wordsCache = LruCache<Uri, List<String>>(4)
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

Then, our revised `generate()` function gets the cached word list by examining our cache using the supplied `Uri` and proceeds from there:

```
ListenableFuture<List<String>> generate(Uri wordsDoc, int wordCount) {
    return CallbackToFutureAdapter.getFuture(completer -> {
        threadPool.execute(() -> {
```

```
List<String> words;

synchronized (wordsCache) {
    words = wordsCache.get(wordsDoc);
}

try {
    if (words == null) {
        InputStream in;

        if (wordsDoc.equals(ASSET_URI)) {
            in = assets.open(PassphraseRepository.ASSET_FILENAME);
        }
        else {
            in = resolver.openInputStream(wordsDoc);
        }

        words = readWords(in);
        in.close();

        synchronized (wordsCache) {
            wordsCache.put(wordsDoc, words);
        }
    }

    completer.set(rollDemBones(words, wordCount));
}
catch (Throwable t) {
    completer.setException(t);
}
});

return "generate words";
});
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

```
suspend fun generate(
    context: Context,
    wordsDoc: Uri,
    count: Int
): List<String> {
    var words: List<String>?

    synchronized(wordsCache) {
        words = wordsCache.get(wordsDoc)
    }
}
```

```
return words?.let { rollDemBones(it, count, random) }
    ?: loadAndGenerate(context, wordsDoc, count)
}

private suspend fun loadAndGenerate(
    context: Context,
    wordsDoc: Uri,
    count: Int
): List<String> = withContext(Dispatchers.IO) {
    val inputStream: InputStream? = if (wordsDoc == ASSET_URI) {
        context.assets.open(ASSET_FILENAME)
    } else {
        context.contentResolver.openInputStream(wordsDoc)
    }

    inputStream?.use {
        val words = it.readLines()
            .map { line -> line.split("\t") }
            .filter { pieces -> pieces.size == 2 }
            .map { pieces -> pieces[1] }

        synchronized(wordsCache) {
            wordsCache.put(wordsDoc, words)
        }

        rollDemBones(words, count, random)
    } ?: throw IllegalStateException("could not open $wordsDoc")
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

If we do not have our words yet, we need to load them. In `DiceLight`, we would always open the asset. Now, we see if the asset is the magic `ASSET_URI`, and we only use `AssetManager` and its `open()` function if that is the case. Otherwise, we use a `ContentResolver` and `openInputStream()` to get the content identified by the `Uri` that we got from `ACTION_OPEN_DOCUMENT`. You can get a `ContentResolver` by calling `getContentResolver()` on a `Context`, such as the one supplied to the repository constructor (Java) or `generate()` function (Kotlin).

The rest of `PassphraseRepository` is the same, as the rest of the code for loading words does not care whether the `InputStream` came from an asset or came from the `ContentResolver`.

DocumentFile and the Rest of the CRUD

`ACTION_OPEN_DOCUMENT` (and `ActivityResultContracts.OpenDocument`) will give you a `Uri` for a document that you can open for reading — the “R” in “CRUD”, as we saw in *Diceware*. The Storage Access Framework also supports the remaining operations: create, update, and delete.

To help you with these operations, the Jetpack offers a `DocumentFile` class, which provides convenience functions for finding out key details about the `Uri` that you received. For `ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT`, you can get a `DocumentFile` for a `Uri` by calling `DocumentFile.fromSingleUri()`, passing in that `Uri`. `DocumentFile` then has functions like `getType()` to tell you the MIME type associated with that particular piece of content.

Create

`ACTION_CREATE_DOCUMENT` will give you a `Uri` for a document that you can open for writing, as it is your document. Nowadays, you would use `ActivityResultContracts.CreateDocument()` for this.

`ACTION_CREATE_DOCUMENT` supports an extra, named `EXTRA_TITLE`, containing your desired filename or other “display name” — this does not have to be a classic filename with an extension. The `launch()` function for `ActivityResultContracts.CreateDocument()` takes a string that serves in the same role.

Note, though, that the user has the right to replace your proposed title with something else. You can find out the title for a `Uri` by calling `getName()` on a `DocumentFile`. Again, bear in mind that this does not have to have a classic filename structure. In particular, it does not need to have a file extension.

Update

The `Uri` returned from an `ACTION_OPEN_DOCUMENT` request may be writable; a `Uri` from `ACTION_CREATE_DOCUMENT` should be writable. You can find out by calling `canWrite()` on a `DocumentFile` for the `Uri`. If that returns `true`, you can use `openOutputStream()` on a `ContentResolver` to write to that document.

Delete

If you can write to the content, you can also delete it. To do that, call `delete()` on a `DocumentFile` for that `Uri`.

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 another activity — 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. Note that fragments are all considered to be a part of the activity that created them, so you do not need to worry about extending rights from one fragment to another.

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. Then, you can save the `Uri` somewhere — such as in `SharedPreferences`, which we will explore [in an upcoming chapter](#). Later, when your app runs again, you can get the `Uri` and probably still use it with `ContentResolver` and `DocumentFile`, even for a completely new activity or completely new process. Those rights even survive a reboot.

However, those rights will not survive the document being deleted or moved by some other app. You can call `exists()` on a `DocumentFile` to see if your `Uri` still points to a document that exists.

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.

Document Trees

`ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT` are sufficient for most apps.

However, there may be cases where you need the equivalent of a “choose directory”

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 “directory”, into which you can create all of those individual bits of content.

For that, the Storage Access Framework offers document trees.

Getting a Tree

Instead of using `ACTION_OPEN_DOCUMENT`, you can use `ACTION_OPEN_DOCUMENT_TREE...` or, nowadays, `ActivityResultContracts.OpenDocumentTree`. You will get back a `Uri` that represents the tree. You should have full read/write access not only to this tree but to anything inside of it.

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.

Android 11+ Restrictions

Since the beginning, the Storage Access Framework has been billed as the way for the app to get access to whatever content the user wants to work with.

In Android 11+, that is no longer the case, as the OS will prevent the user from accessing the user's content in scenarios that Google does not like.

Principally, this affects `ACTION_OPEN_DOCUMENT_TREE`, preventing the user from selecting:

- The root of external storage
- The `Download/` directory
- Any subdirectories off of the `Android/` directory on external or removable storage

`ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT` share that last restriction of `ACTION_OPEN_DOCUMENT_TREE`. Otherwise, these actions seem unaffected.

Unfortunately, that `ACTION_OPEN_DOCUMENT` limitation means that an app's files are inaccessible by the user except through that app or by copying the files elsewhere using a device-supplied file manager.

Using Preferences

Android allows apps to keep preferences, in the form of key/value pairs (akin to a Map), that will persist 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 rudimentary value (boolean, String, etc.)

The principal use of SharedPreferences, though, is for storing data that is collected by a preference UI. Android provides a system for producing a UI that looks a lot like the individual screens of the Settings app. That UI is fairly easy to set up and is a recommended solution for configuration options and related settings.

In this chapter, we will examine how to set up a preference UI and work with SharedPreferences

The Preferred Preferences

The good news is that there is only one SharedPreferences class.

The bad news is that there are a few implementations of the preference UI. The native one — classes in `android.preference` — has been deprecated. Instead, we are supposed to use the AndroidX implementation.

The SimplePrefs sample module in the [Sampler](#) and [SamplerJ](#) projects use the AndroidX implementation, in the form of the `androidx.preference:preference` library. This sample app offers a preference UI (`PrefsFragment`) and a regular UI that shows the current value of the user-supplied preferences (`HomeFragment`).

Collecting Preferences with PreferenceFragmentCompat

Some “preferences” will be collected as part of the natural use of your user interface. For example, if you have a SeekBar widget 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. For that, typically we use preference XML resources and a PreferenceFragmentCompat.

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, you will define an XML resource file that describes the preferences that you want. The root element of this XML file will be `<PreferenceScreen>`, and it will contain child elements, generally 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/checkSummary"
        android:title="@string/checkTitle"/>

    <EditTextPreference
        android:dialogTitle="@string/dialogTitle"
        android:key="field"
        android:summary="@string/fieldSummary"
        android:title="@string/fieldTitle"/>

    <ListPreference
        android:dialogTitle="@string/listDialogTitle"
        android:entries="@array/cities"
```

USING PREFERENCES

```
android:entryValues="@array/airportCodes"
android:key="list"
android:summary="@string/listSummary"
android:title="@string/listTitle"/>

</PreferenceScreen>
```

(from [SimplePrefs/src/main/res/xml/preferences.xml](#))

If you open up that resource in Android Studio, you will be given an editor that is reminiscent of the layout resource editor, with XML and graphical editors.

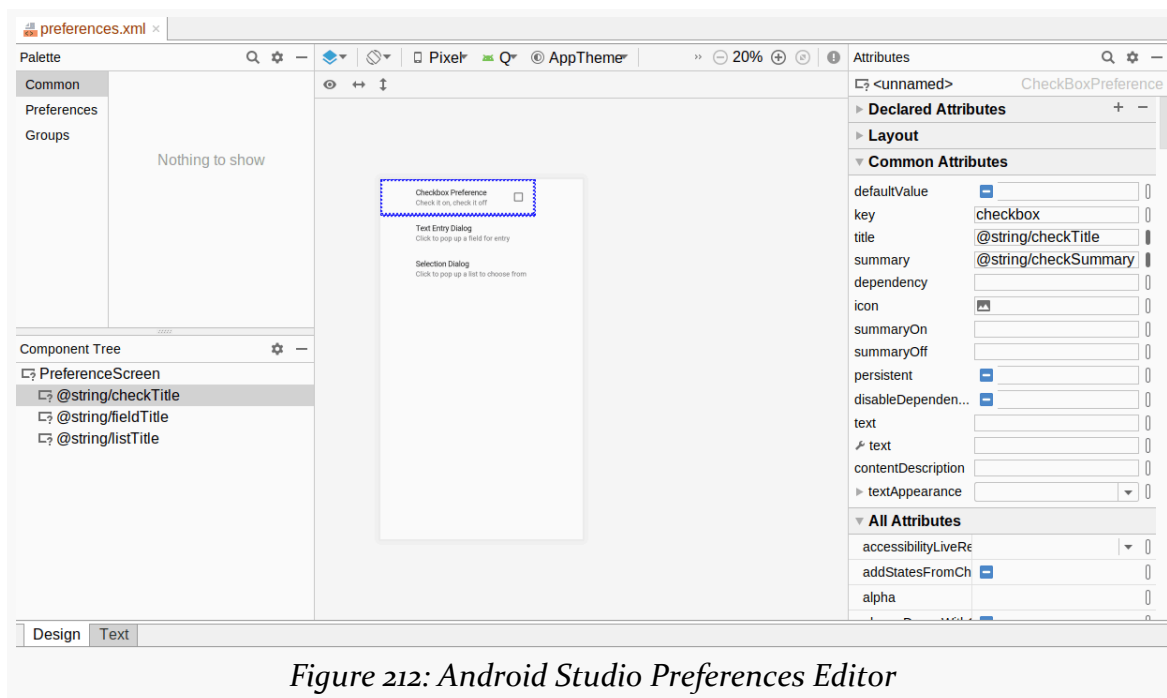


Figure 212: Android Studio Preferences Editor

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”.

As with widgets in a layout resource, the element names of the preferences reflect a Java class that is the implementation of that preference. Our preference XML has `CheckBoxPreference`, `EditTextPreference`, and `ListPreference` elements, so our UI will be constructed from those classes. Note that these are *not* widgets — they do

not extend from View — so you cannot use them directly in a layout resource.

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 these preference elements and others like them [later in this chapter](#).

Creating Your Preference Fragment

We use preference XML resources with a `PreferenceFragmentCompat` class. This is a type of fragment that knows:

- How to load preference XML, inflating it into the actual Java objects
- How to render the UI for those preferences
- How to populate that UI with the current preference values
- How to update the preference values based on user input

Typically, a `PreferenceFragmentCompat` subclass just overrides `onCreatePreferences()` and calls `addPreferencesFromResource()`:

```
package com.commonware.jetpack.simpleprefs;

import android.os.Bundle;
import androidx.preference.PreferenceFragmentCompat;

public class PrefsFragment extends PreferenceFragmentCompat {
    @Override
    public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

(from [SimplePrefs/src/main/java/com/commonware/jetpack/simpleprefs/PrefsFragment.java](#))

USING PREFERENCES

```
package com.commonware.jetpack.simpleprefs

import android.os.Bundle
import androidx.preference.PreferenceFragmentCompat

class PrefsFragment : PreferenceFragmentCompat() {
    override fun onCreatePreferences(
        savedInstanceState: Bundle?,
        rootKey: String?
    ) {
        addPreferencesFromResource(R.xml.preferences)
    }
}
```

(from [SimplePrefs/src/main/java/com/commonware/jetpack/simpleprefs/PrefsFragment.kt](#))

Otherwise, this is an ordinary Fragment. We can start it using a `FragmentManager` or the Navigation component, as we see fit. In this sample, we use the Navigation component, linking a `HomeFragment` to the `PrefsFragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation android:id="@+id/nav_graph"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:label="@string/app_name"
    app:startDestination="@id/homeFragment">

    <fragment
        android:id="@+id/homeFragment"
        android:name="com.commonware.jetpack.simpleprefs.HomeFragment"
        android:label="@string/app_name">
        <action
            android:id="@+id/editPrefs"
            app:destination="@id/prefsFragment" />
        </fragment>
    <fragment
        android:id="@+id/prefsFragment"
        android:name="com.commonware.jetpack.simpleprefs.PrefsFragment"
        android:label="@string/app_name" />
</navigation>
```

(from [SimplePrefs/src/main/res/navigation/nav_graph.xml](#))

The UI

If you run the app and click the “Edit Preferences” button, you will be taken to `PrefsFragment` and the UI that it creates:

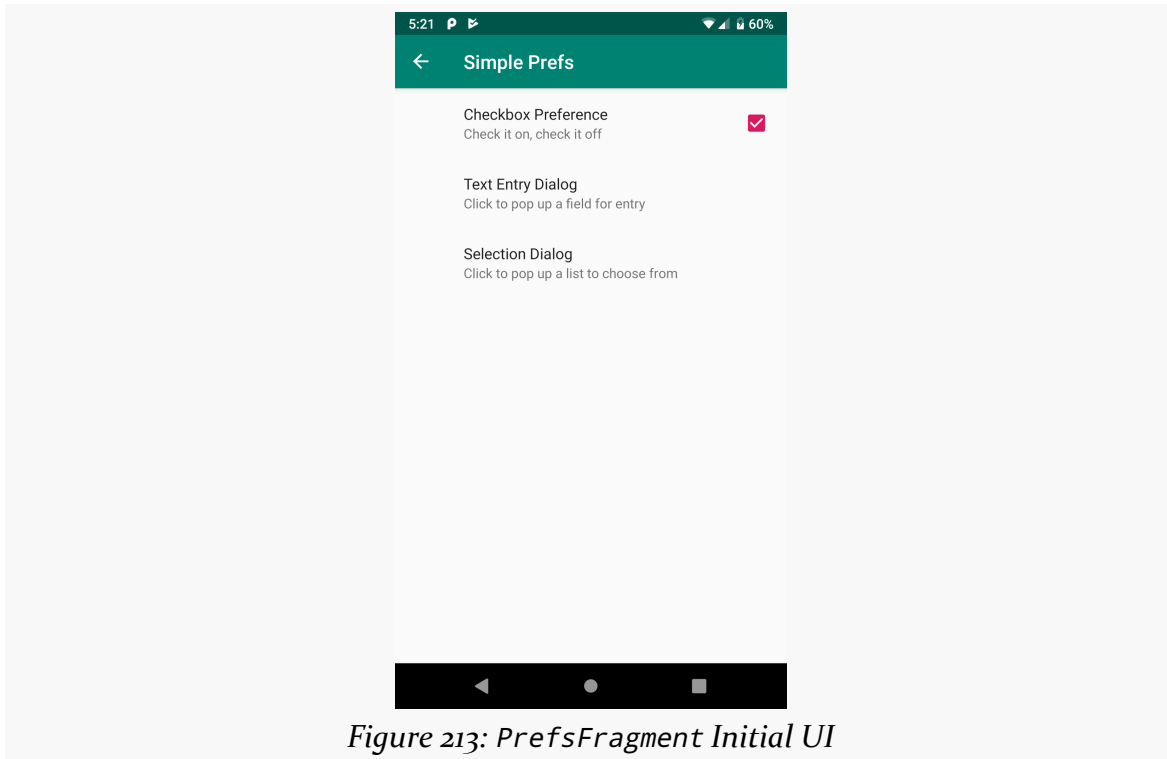


Figure 213: PrefsFragment Initial UI

`CheckBoxPreference` is an “inline” preference: the user can set the value from a widget right in the preference itself. In the case of `CheckBoxPreference`, that is in the form of a `CheckBox` widget.

USING PREFERENCES

Our other two preferences are dialog preferences, where the user taps on the preference to bring up a dialog where the user sets the value:

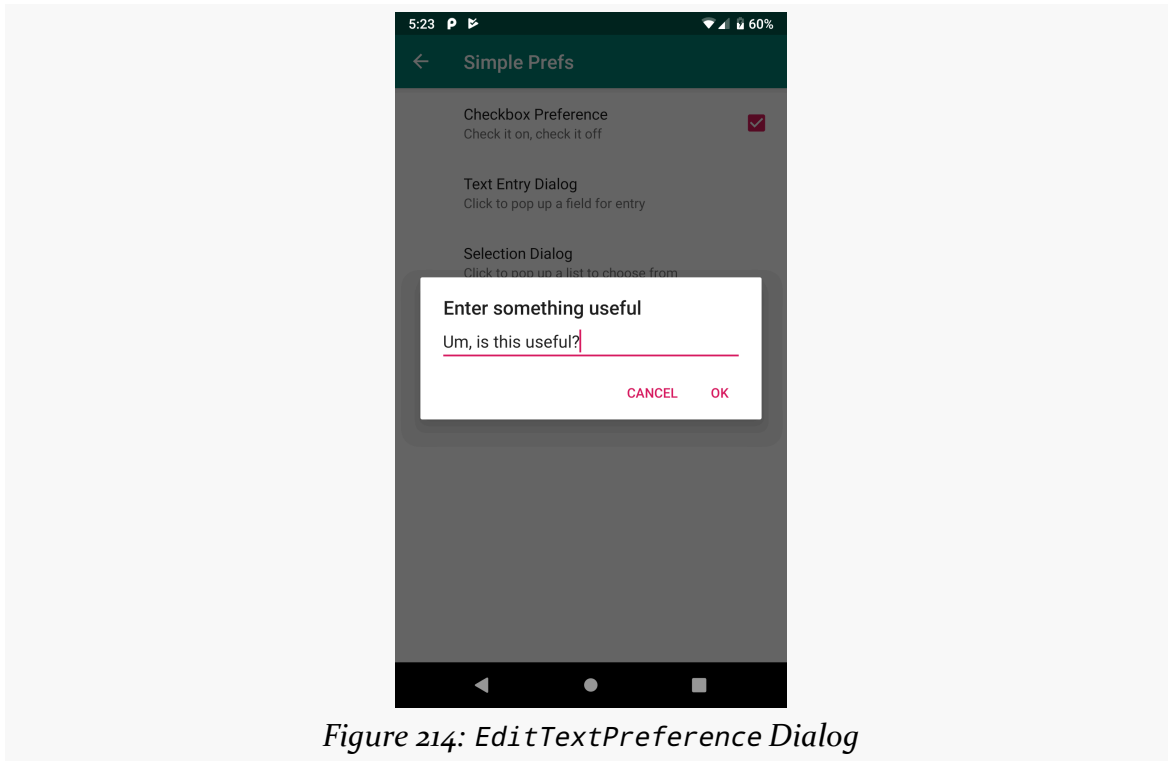


Figure 214: EditTextPreference Dialog

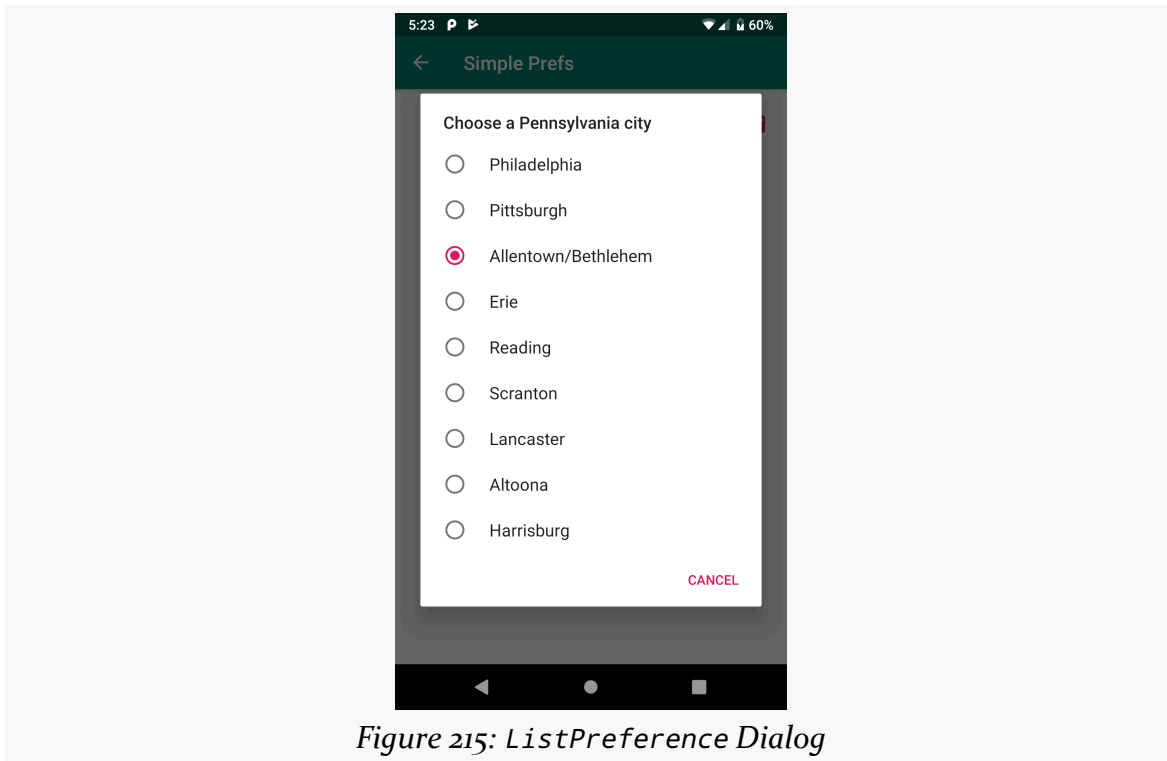


Figure 215: ListPreference Dialog

Each value is saved in the `SharedPreferences` as the user changes it. There is no “save” action that the user needs to do in order to save the changes.

Types of Preferences

The elements in the preference XML will refer to subclasses of `androidx.preference.Preference`. There are several of these in the current AndroidX Preference library, and most may be added in the future. You can also create your own, such as by extending `DialogPreference` and calling various methods to build up the content of the dialog (e.g., `setDialogTitle()`, `setDialogLayoutResource()`).

CheckBoxPreference and SwitchPreference

The sample application shown above a `CheckBoxPreference`. As noted, 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.

The value that will be stored in the SharedPreferences is a boolean — we will explore how to read and manipulate these values from your own code [later in this chapter](#).

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`. Also, as the sample app shows, you can have `android:dialogTitle` to provide the title for the dialog that wraps the EditText widget.

The value stored in the SharedPreferences is a string.

ListPreference and MultiSelectListPreference

A ListPreference displays a dialog with your choice of entries. Each is accompanied by a RadioButton, with the checked RadioButton indicating the current value (if any). A MultiSelectListPreference has the same look, except it has checkboxes for each entry, and the user can choose multiple values, not just one.

To configure what appears in the list, you provide two attributes in the ListPreference or MultiSelectListPreference element:

- `android:entries` provides what the user sees
- `android:entryValues` provides the corresponding values that are saved in the SharedPreferences

In the preference XML, these attributes need to point to string-array resources. 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.

USING PREFERENCES

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="airportCodes">
    <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 [SimplePrefs/src/main/res/values/arrays.xml](#))

Here, the actual strings are written in-line. They could just as easily be references to string resource. 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/airportCodes"
  android:key="list"
  android:summary="@string/listSummary"
  android:title="@string/listTitle"/>
```

(from [SimplePrefs/src/main/res/xml/preferences.xml](#))

The dialog will show the strings in the `android:entries` array. The value that matches, position-wise, from the `android:entryValues` array is what gets saved in the `SharedPreferences`. So, if the user chooses Pittsburgh as the city, `PIT` will be the value saved in the `SharedPreferences`. If you want the user-visible strings to be the same as what goes into the `SharedPreferences`, just use the same string-array resource for both `android:entries` and `android:entryValues`.

A `ListPreference` saves a string to `SharedPreferences`. A `MultiSelectListPreference` saves a Set of strings to the `SharedPreferences`.

There is also `DropDownPreference`, which works like a `ListPreference` but uses a drop-down list presentation, rather than a pop-up dialog.

SeekBarPreference

A `SeekBarPreference` shows a `SeekBar` widget, to allow the user to specify a value in a range. It saves its value as an `int` to the `SharedPreferences`.

Working with SharedPreferences

At some point, our app needs to use the values that the user provided via the preference UI. We might also want to save other data in `SharedPreferences` that is not part of the preference UI.

Reading Preferences

Call `getDefaultSharedPreferences()` on `android.preference.PreferenceManager` to get the `SharedPreferences` that is used by the preference UI for its values.

`SharedPreferences` 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.

You can read values at any point. If you want to find out when the preference UI (or other code in your app) changes the preferences, call `registerOnSharedPreferenceChangeListener()` on the `SharedPreferences` to register an `OnSharedPreferenceChangeListener` to be notified of when values change.

USING PREFERENCES

The sample app has a `ConstraintLayout` that shows the values of our three preferences:

```
<?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"
    xmlns:tools="http://schemas.android.com/tools">

    <data>

        <variable
            name="state"
            type="androidx.lifecycle.LiveData<com.commonsware.jetpack.simpleprefs.HomeViewState>" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".HomeFragment">

        <TextView
            android:id="@+id/checkLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="8dp"
            android:text="@string/labelCheck"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/fieldLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="8dp"
            android:text="@string/fieldLabel"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/checkLabel" />

        <TextView
            android:id="@+id/listLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="8dp"
            android:text="@string/listLabel"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/fieldLabel" />

        <androidx.constraintlayout.widget.Barrier
            android:id="@+id/barrier"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:barrierDirection="end"
            app:constraint_referenced_ids="checkLabel,fieldLabel,listLabel" />

        <TextView
```

USING PREFERENCES

```
    android:id="@+id/checkValue"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:text="@{state.isChecked ? @string/checked : @string/unchecked}"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    app:layout_constraintBaseline_toBaselineOf="@id/checkLabel"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/barrier"
    tools:text="checked" />

<TextView
    android:id="@+id/fieldValue"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:text="@{state.fieldValue}"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    app:layout_constraintBaseline_toBaselineOf="@id/fieldLabel"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/barrier"
    tools:text="Something" />

<TextView
    android:id="@+id/listValue"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:text="@{state.listView}"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    app:layout_constraintBaseline_toBaselineOf="@id/listLabel"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/barrier"
    tools:text="ABE" />

<Button
    android:id="@+id/edit"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@string/editPrefs"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/listValue" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [SimplePrefs/src/main/res/layout/fragment_home.xml](#))

We are using data binding to populate three of the TextView widgets with certain values, pulled from a HomeViewState that we get via LiveData. That comes from HomeMotor implementation of AndroidViewModel:

```
package com.commonware.jetpack.simpleprefs;

import android.app.Application;
```


USING PREFERENCES

```
import android.content.SharedPreferences;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.preference.PreferenceManager;

public class HomeMotor extends AndroidViewModel {
    private final SharedPreferences prefs;
    private final MutableLiveData<HomeViewState> states = new MutableLiveData<>();

    public HomeMotor(@NonNull Application application) {
        super(application);

        prefs = PreferenceManager.getDefaultSharedPreferences(application);
        prefs.registerOnSharedPreferenceChangeListener(LISTENER);
        emitState();
    }

    @Override
    protected void onCleared() {
        prefs.unregisterOnSharedPreferenceChangeListener(LISTENER);
    }

    LiveData<HomeViewState> getStates() {
        return states;
    }

    private void emitState() {
        states.setValue(
            new HomeViewState(prefs.getBoolean("checkbox", false),
                prefs.getString("field", ""), prefs.getString("list", "")));
    }

    private SharedPreferences.OnSharedPreferenceChangeListener LISTENER =
        (prefs, key) -> emitState();
}
```

(from [SimplePrefs/src/main/java/com/commonsware/jetpack/simpleprefs/HomeMotor.java](#))

```
package com.commonsware.jetpack.simpleprefs

import android.app.Application
import android.content.SharedPreferences
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.preference.PreferenceManager

class HomeMotor(application: Application) : AndroidViewModel(application) {
    private val listener =
        SharedPreferences.OnSharedPreferenceChangeListener { _, _ -> emitState() }
    private val prefs = PreferenceManager.getDefaultSharedPreferences(application)
    private val _states = MutableLiveData<HomeViewState>()
    val states: LiveData<HomeViewState> = _states

    init {
        prefs.registerOnSharedPreferenceChangeListener(listener)
        emitState()
    }
}
```

USING PREFERENCES

```
override fun onCleared() {
    prefs.unregisterOnSharedPreferenceChangeListener(listener)
}

private fun emitState() {
    _states.value = HomeViewState(
        prefs.getBoolean("checkbox", false),
        prefs.getString("field", "") ?: "",
        prefs.getString("list", "") ?: ""
    )
}
}
```

(from [SimplePrefs/src/main/java/com/commonsware/jetpack/simpleprefs/HomeMotor.kt](https://github.com/commonsware/jetpack/simpleprefs/blob/master/src/main/java/com/commonsware/jetpack/simpleprefs/HomeMotor.kt))

When the motor is created, we call `PreferenceManager.getDefaultSharedPreferences()` to retrieve the `SharedPreferences` object. Then, we call `registerOnSharedPreferenceChangeListener()` to be notified of changes, then call `emitState()`. That reads the values of our three preferences (using `getBoolean()` and `getString()` methods) and puts them in the `HomeViewState` for the UI to use. Later on, when the `HomeMotor` is cleared, we call `unregisterOnSharedPreferenceChangeListener()`, so we no longer get updates (and so the motor can be garbage-collected).

USING PREFERENCES

The net result is that the HomeFragment will show the initial default values, then will reflect the results of any changes that you make:

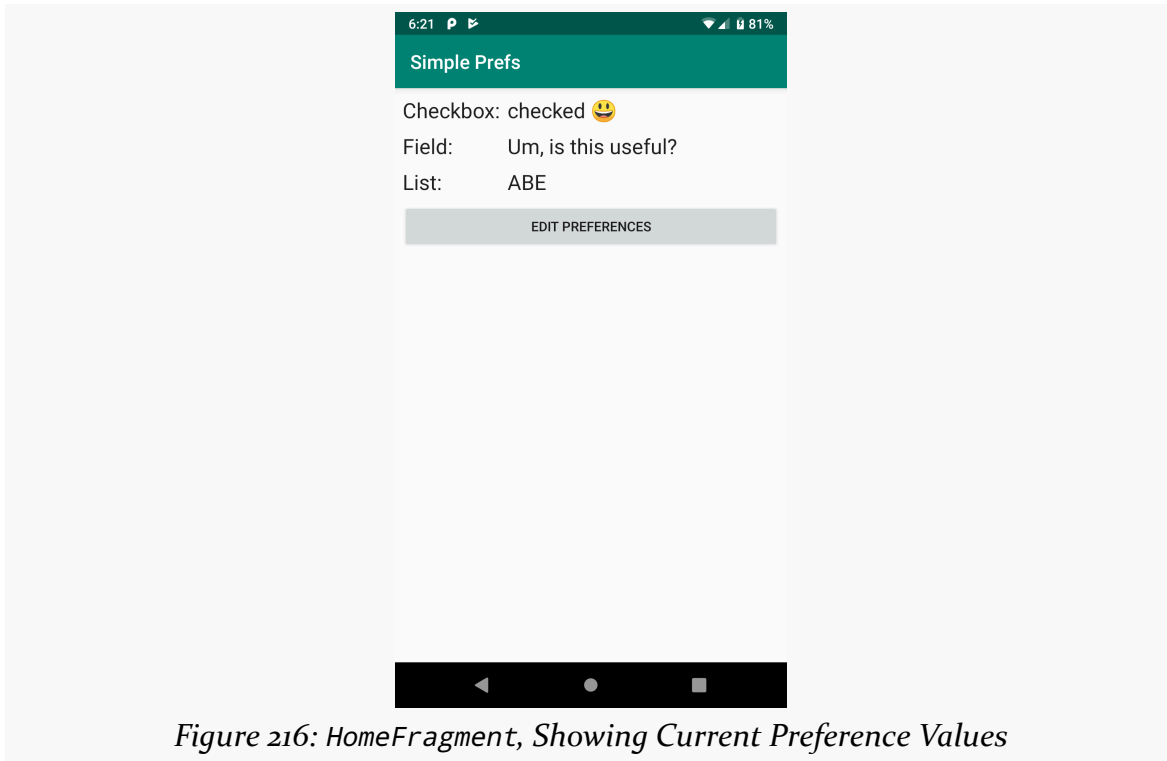


Figure 216: HomeFragment, Showing Current Preference Values

Modifying Preferences

Call `edit()` on the `SharedPreferences` object 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. If you are on a background thread already, call `commit()`; otherwise, call `apply()`.

Requesting Permissions

Some things that your app might want to do — access the Internet, get a GPS fix, get personal information about the user’s contacts, etc. — will require permission from the user. Sometimes the user grants permission implicitly, just by installing your app. Sometimes, the user grants permission explicitly, for things that Google considers to be “dangerous”. And your app will need to deal with these permissions, including the possibility that the user decides not to grant you a permission.

In this chapter, we will explore the basics of Android’s permission system.

Frequently-Asked Questions About Permissions

Permissions are frequently a confusing topic in Android app development, particularly with respect to those “dangerous” permissions. So, here is a FAQ about Android’s permissions system.

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, in the form of a `<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 more common ones:

- `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
- `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE`, for working with files directly on [external storage](#)
- `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, for determining where the device is, such as via GPS
- `READ_CONTACTS`, to get at personally-identifying information of arbitrary contacts that the user has in their Contacts app
- `BLUETOOTH` and `BLUETOOTH_ADMIN`, for communicating with other devices over Bluetooth
- `CAMERA`, for taking pictures directly using the camera APIs or wrapper libraries like the CameraX library
- `RECEIVE_BOOT_COMPLETED`, to get control when the device reboots

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 a framework 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.

What Are “Normal” and “Dangerous” Permissions?

In modern versions of Android, a normal permission is one that we have to request via the manifest, but the user does not need to approve explicitly. Just by installing the app, they implicitly grant us the permission. They can find out about these permissions (e.g., via the Play Store listing), but they cannot deny them, other than by uninstalling the app. Generally, normal permissions are ones that have limited user impact (e.g., `RECEIVE_BOOT_COMPLETED`, which only affects behavior on a reboot). The biggest exception to that rule is the `INTERNET` permission, which is a normal permission because a *lot* of apps want to be able to access the Internet.

By contrast, a dangerous permission is one that the user must approve explicitly. On Android 5.1 and older devices, this occurs at install time, either on the device or in the Play Store (for apps distributed through that channel). On Android 6.0+ devices, not only must our app have a `<uses-permission>` element in the manifest, but we also need to call methods to request that permission at runtime, at the point when we first need the permission. Mostly, dangerous permissions are ones that have privacy or security ramifications.

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" encoding="utf-8"?>
<manifest package="com.commonware.jetpack.contenteditor"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

  <application
    android:allowBackup="true"
    android:requestLegacyExternalStorage="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
```

REQUESTING PERMISSIONS

```
    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>
```

(from [ContentEditor/src/main/AndroidManifest.xml](#))

For normal permissions, this is all that you need. For dangerous permissions — such as `WRITE_EXTERNAL_STORAGE` — there is more work to be done, as we will explore [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. If you look at the “Merged Manifest” sub-tab of the manifest editor, you will see all of the `<uses-permission>` elements that will be included in your app and where they come from.

Dangerous Permissions: Request at Runtime

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.

Let’s explore the runtime permissions system via a new series of questions.

Along the way, we will examine bits of the `ContentEditor` sample module in the [Sampler](#) and [SamplerJ](#) projects. This app implements a tiny text editor, where the user can edit text from various sources. We will explore the file I/O portions of this code in [an upcoming chapter](#). Here, we will focus on the `WRITE_EXTERNAL_STORAGE` permission that this app needs, as that is a dangerous permission and one that we have to request at runtime.

What Permissions Are Affected By This?

Inside Android, permissions are organized into permission groups. For example, in Android 9.0, there are ten permission groups that contain dangerous permissions:

Permission Group	Permission
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
CALL_LOG	PROCESS_OUTGOING_CALLS, READ_CALL_LOG, WRITE_CALL_LOG
CAMERA	CAMERA
CONTACTS	GET_ACCOUNTS, READ_CONTACTS, WRITE_CONTACTS
LOCATION	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	ADD_VOICEMAIL, ANSWER_PHONE_CALLS, CALL_PHONE, READ_PHONE_NUMBERS, READ_PHONE_STATE, USE_SIP
SENSORS	BODY_SENSORS
SMS	READ_SMS, RECEIVE_SMS, RECEIVE_MMS, RECEIVE_WAP_PUSH, SEND_SMS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE

This roster changes over time. For example, in Android 6.0, the call log permissions were in the PHONE category.

REQUESTING PERMISSIONS

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:

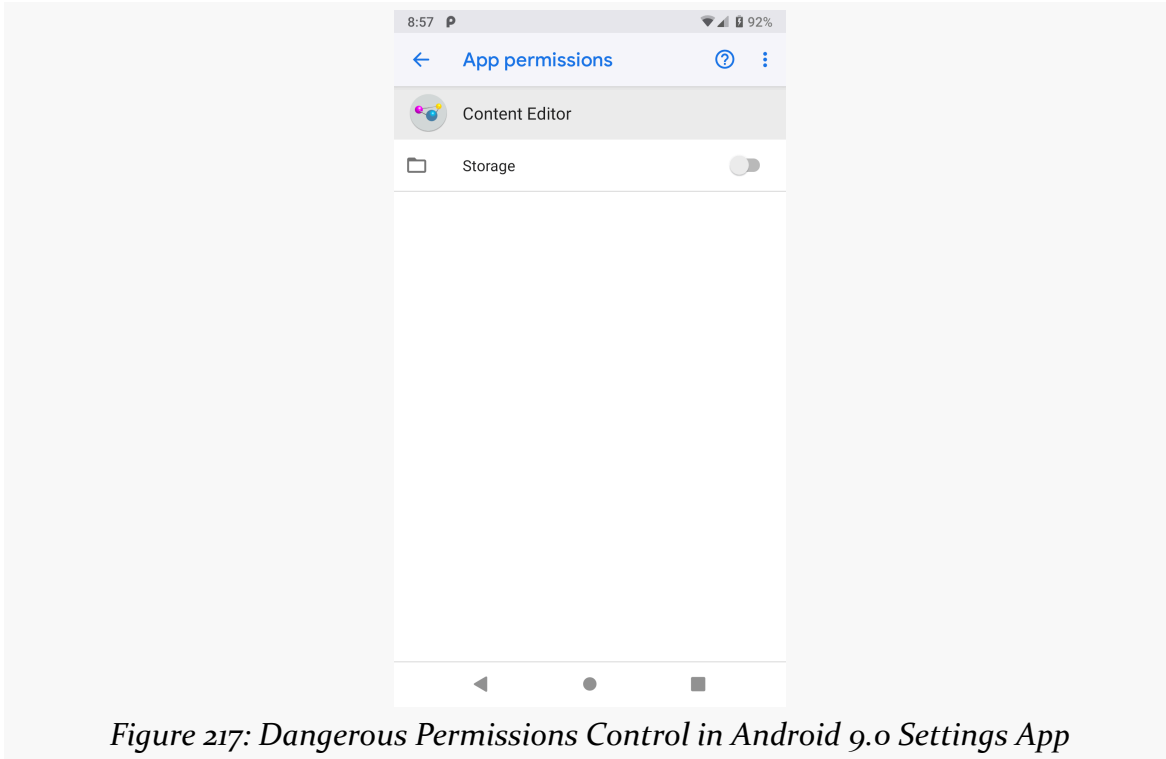


Figure 217: Dangerous Permissions Control in Android 9.0 Settings App

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 Ask the User For Permission?

To ask the user for one of the runtime permissions — or find out that we already have it — we can use `ActivityResultContracts.RequestPermission` along with `registerForActivityResult()`:

```
private final ActivityResultLauncher<String> requestPerm =
    registerForActivityResult(new ActivityResultContracts.RequestPermission(),
        new ActivityResultCallback<Boolean>() {
            @Override
```

REQUESTING PERMISSIONS

```
public void onActivityResult(Boolean wasGranted) {
    if (wasGranted) {
        loadFromDir(Environment.getExternalStorageDirectory());
    }
    else {
        Toast.makeText(MainActivity.this, R.string.msg_sorry, Toast.LENGTH_LONG).show();
    }
}
});
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
private val requestPerm = registerForActivityResult(ActivityResultContracts.RequestPermission()) {
    if (it) {
        loadFromDir(Environment.getExternalStorageDirectory())
    } else {
        Toast.makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG).show()
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

We can then call `launch()` on `requestPerm` when we want to request permission, supplying the name of the permission that we want to request (`Manifest.permission.WRITE_EXTERNAL_STORAGE`):

```
requestPerm.launch(Manifest.permission.WRITE_EXTERNAL_STORAGE);
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
requestPerm.launch(Manifest.permission.WRITE_EXTERNAL_STORAGE)
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

The callback will be called with a Boolean value. If it is true, we received permission; if false, we did not. Based upon that value, we can determine what we want to do.

What Do I Do If the User Says “No”?

The user can click the “Deny” button for one or more pages in that system dialog and thereby reject granting you the requested permission(s). Your callback will get false in this case, instead of true.

If you were requesting permission as a direct response to some bit of user input (e.g., user tapped on an app 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

REQUESTING PERMISSIONS

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 app bar items).

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 “Don’t ask again” option:

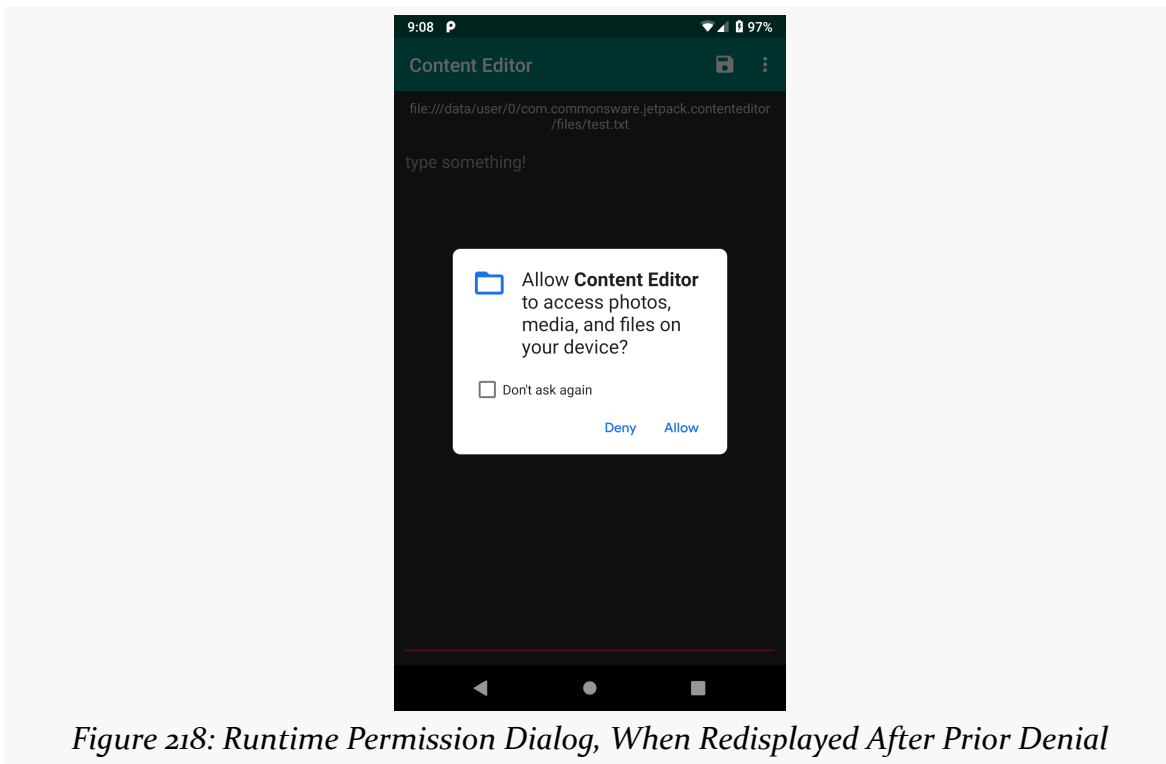


Figure 218: Runtime Permission Dialog, When Redisplayed After Prior Denial

If the user chooses that, not only will you not get the runtime permission now, but all future requests will immediately call your callback 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:

REQUESTING PERMISSIONS

- Disabling UI input (e.g., app 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 count 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.

How Do I Know If the User Takes Permissions Away From Me?

Permissions granted by a user can be revoked in a few ways:

- The user can remove it manually via the Settings app
- On Android 11+, the user can choose to grant a one-time permission, which your app will only hold until it leaves the foreground
- On Android 11+, the user can opt into having Android revoke the permission automatically if the user does not use your app for some time

If a permission is revoked, and your app is running at the time, 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.

What Happens When I Ship This to an Android 5.1 or 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 using `ActivityResultContracts.RequestPermission` as described above,

your code should just work.

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, and you will need to request the permissions.

Handling Files

Most programmers, early on, learn how to read and write files. File I/O has been a staple of computer programming for decades. And, on Android, you can read and write files, much as you can with other operating systems.

The biggest difference in Android is *where* you can read and write files. That is quite a bit different than what you may be used to. And, increasingly, Google is pushing developers away from files entirely, steering us in the direction of the Storage Access Framework, as we saw in [the chapter on content](#).

In this chapter, we will explore where we can read and write files.

The Three Types of File Storage

File storage locations in Android break down into three main types: internal, external, and removable.

Internal Storage

A user will think that “internal storage” refers to what they get when they plug their phone into a desktop via a USB cable. From the standpoint of the Android SDK, though, that is really external storage.

Internal storage, from the Android SDK’s perspective, refers to portions of the on-board flash that are both:

- Private to your app, and
- Invisible to the user

The only way that the user will interact with internal storage is through your app.

We have already worked a bit with internal storage: `SharedPreferences` are on internal storage. Later, when we [examine Room](#), its databases are stored on internal storage by default. In terms of standard Java/Kotlin file I/O code, you find your internal storage locations via methods on `Context`. The two most common of those methods are:

- `getFilesDir()`, which returns a place where you can read and write files, create subdirectories, etc.
- `getCacheDir()`, which returns a separate directory where you can also read and write files (and so on)... but where the system might delete your files to free up space for the user

External Storage

What the user thinks is “internal storage” is really “external storage” in the terms used by the Android SDK documentation. There are two main sets of locations on external storage that you can use: app-specific locations, and the overall shared portions of external storage.

App-Specific

There are methods on `Context` named `getExternalFilesDir()` and `getExternalCacheDir()`. These return locations on external storage that are unique for your app. And, as with their `getFilesDir()/getCacheDir()` counterparts, the “cache” ones are eligible to be cleared by the OS to free up disk space.

Unlike `getFilesDir()` and `getCacheDir()`, though, all locations on external storage can be accessed by the user and may be able to be accessed by other apps on the device. However, `getExternalFilesDir()` and `getExternalCacheDir()` are locations that are *unique* for your app — if you put things there, other apps should not accidentally overwrite or otherwise modify them.

Also, unlike `getFilesDir()` and `getCacheDir()`, `getExternalFilesDir()` and `getExternalCacheDir()` take a parameter. Typically, you pass in `null`.

Shared

Apps running on Android 9.0 and older can work with external storage overall via

methods on the `Environment` class. In particular:

- `Environment.getExternalStorageDirectory()` returns the root of external storage
- `Environment.getExternalStoragePublicDirectory()` returns a specific common location on external storage, based on a supplied parameter (e.g., `Environment.DIRECTORY_DOWNLOADS` for a Downloads/ directory)

These locations are visible to the user and may be visible to other apps. And, since they are common and shared, other apps are more likely to manipulate files that you place here. At the same time, you may be able to manipulate the files of other apps, or files placed here by the user.

On newer versions of Android, these methods are deprecated, and you will not have access to them by default. Google is trying to steer you to the Storage Access Framework as an alternative.

Those location identifiers on `Environment`, such as `DIRECTORY_DOWNLOADS`, can also be passed to `getExternalFilesDir()` and `getExternalCacheDir()`, to get access to an app-specific directory on external storage for that type of material.

Removable Storage

Removable storage refers to micro SD cards, USB thumb drives, or anything else that the device supports that can be physically removed by the user, without breaking their phone.

For several years, Android had no official support for removable storage. Nowadays, not only does the Storage Access Framework work with it, but there are methods on `Context` that let you get to app-specific directories on removable storage. Calling `getExternalFilesDirs()`, `getExternalCacheDirs()`, or `getExternalMediaDirs()` (note the plural method names) will return an array of `File` objects representing directories. The first element in that array will be a location on external storage – in the case of `getExternalFilesDirs()` and `getExternalCacheDirs()`, it should be the same location as you get from calling `getExternalFilesDir()` and `getExternalCacheDir()`. If the array has 2+ elements, all but the first will point to an app-specific directory on some removable medium. You can read and write files here, create subdirectories, etc.

What the User Sees

On Android devices:

- Users cannot see any of your files on internal storage, unless they root their device
- Users can see all of your files on external storage and removable storage

Anything a user can see, a user can read, write, or delete at will.

Note, though, that the user cannot see files on external or removable storage right away through some apps. Apps that rely on a central index of media called `MediaStore` will only find out about your new files when they get indexed. That will happen at some point automatically. You can use `MediaConnection.scanFile()` to force Android to index your new files more quickly — we will see this in action later in the chapter.

Storage, Permissions, and Access

Android 4.4 through 9.0 had a fairly stable set of rules for when you need permissions:

Storage Type	Storage Methods	Permissions Required
internal	methods on Context	none
external	methods on Context	none
external	methods on Environment	<code>READ_EXTERNAL_STORAGE</code> , <code>WRITE_EXTERNAL_STORAGE</code>
removable	methods on Context	none

So, only if you access shared locations on external storage do you need permission. As the names suggest, `READ_EXTERNAL_STORAGE` allows you to read those files, while `WRITE_EXTERNAL_STORAGE` allows you to write those files. These are dangerous permissions, so you need to request them both in the manifest and at runtime, as we saw in [the chapter on permissions](#).

Android 10 changed this. You simply have no filesystem-level access to external or removable storage except by Context methods like `getExternalFilesDir()`. Everything else is locked down by default. To revert this behavior, you can add `android:requestLegacyExternalStorage="true"` to the `<application>` element in your manifest. This attribute will give you the Android 9.0-style behavior on:

- Android 10, and
- Android 11, until you raise your `targetSdkVersion` to 30 or higher

However, on Android 11+, you get *read* access to much of external storage using `READ_EXTERNAL_STORAGE` as before. What you lose is write access. For apps that only need to read from shared locations on external storage, having `android:requestLegacyExternalStorage="true"` means that you will have fairly consistent behavior through the Android versions.

Android 12 — introduced in 2021 — ignores `android:requestLegacyExternalStorage="true"` entirely.

(and, if this all seems overly complicated... it is)

Reading, Writing, and Debugging Storage

Once you have a `File` object that points to a location of interest to you, everything else works like standard Java/Kotlin disk I/O. You can use streams (e.g., `FileInputStream`), readers/writers (e.g., `OutputStreamWriter`), and so on. However, disk I/O may be slow, so you want to do that I/O on a background thread of some form.

For debuggable apps — the sort that you normally run from Android Studio — the IDE will give you somewhat greater ability to view files than ordinary users get. There is a “Device File Explorer” tool in Android Studio, by default docked on the right edge. If you have a device (or emulator) available, it will give you a file explorer for that device that will let you examine the major storage locations of interest to you.

Introducing the Sample App

In [the chapter on permissions](#), we saw a few code snippets from the `ContentEditor` sample module in the [Sampler](#) and [SamplerJ](#) projects. This app implements a tiny text editor, where the user can edit text from various sources.

HANDLING FILES

The app mostly is a large `EditText` widget for typing in some text. Above it is a `TextView` that we use to show a `Uri` representation of the content being shown in the `EditText`. By default, this will open up a file in the app's portion of internal storage:

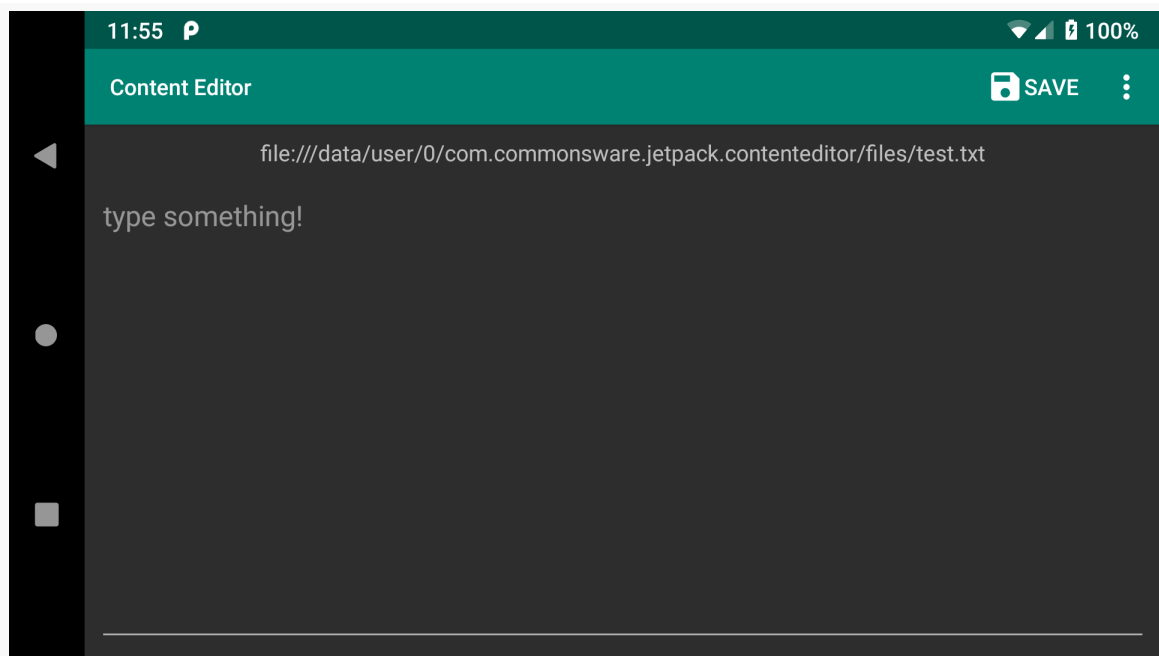


Figure 219: ContentEditor Sample, As Initially Opened

The Save button will let you save any changes that you made to the file. The overflow menu lets you switch the editor to different files or to content that you create using the Storage Access Framework.

Specifying the Location

Our `MainActivity` manages the options menu, including the “Save” item and the overflow. It, therefore, indicates where we should be loading our text from (or saving it to). That is triggered by `onOptionsItemSelected()`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.loadInternal:
            loadFromDir(getFilesDir());
            return true;
    }
}
```

HANDLING FILES

```
case R.id.loadExternal:
    loadFromDir(getExternalFilesDir(null));
    return true;

case R.id.loadExternalRoot:
    requestPerm.launch(Manifest.permission.WRITE_EXTERNAL_STORAGE);
    return true;

case R.id.openDoc:
    openDoc.launch(new String[] { "text/*" });
    return true;

case R.id.newDoc:
    createDoc.launch(FILENAME);
    return true;

case R.id.save:
    motor.write(current, binding.text.getText().toString());
    return true;
}

return super.onOptionsItemSelected(item);
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.loadInternal -> {
            loadFromDir(filesDir)
            return true
        }

        R.id.loadExternal -> {
            loadFromDir(getExternalFilesDir(null))
            return true
        }

        R.id.loadExternalRoot -> {
            requestPerm.launch(Manifest.permission.WRITE_EXTERNAL_STORAGE)
            return true
        }

        R.id.openDoc -> {
            openDoc.launch(arrayOf("text/*"))
            return true
        }
    }
}
```

HANDLING FILES

```
R.id.newDoc -> {
    createDoc.launch(FILENAME)
    return true
}

R.id.save -> {
    current?.let { motor.write(it, binding.text.text.toString()) }
    return true
}
}

return super.onOptionsItemSelected(item)
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

In the case of the `loadInternal` and `loadExternal` items, we call a `loadFromDir()` function providing `getFilesDir()` and `getExternalFilesDir(null)`, respectively. `loadFromDir()` just clears out the `EditText` and passes the location to our `MainMotor` and its `read()` function:

```
private void loadFromDir(File dir) {
    binding.text.setText("");
    motor.read(Uri.fromFile(new File(dir, FILENAME)));
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
private fun loadFromDir(dir: File?) {
    binding.text.setText("")
    motor.read(Uri.fromFile(File(dir, FILENAME)))
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

However, we pass the location to `read()` as a `Uri`, not a `File`. You can use `Uri.fromFile()` to create a `Uri` representation of a file. This is perfectly fine within an app. However, if you try passing such a `Uri` to another app, such as via an `ACTION_VIEW` Intent, you will get a `FileUriExposedException` on Android 7.0+. You can use `FileProvider` to get a safe `Uri` to pass to another app, as we will see [later in this chapter](#).

For the `loadExternalRoot` item, we will wind up calling `loadFromDir()` providing `Environment.getExternalStorageDirectory()` as the location. However, this requires permission from the user. So, in addition to having the `<uses-permission>` element in the manifest, we need to check for this permission at runtime, asking for

HANDLING FILES

it if we do not have it already, as we saw in [the chapter on permissions](#), using `ActivityResultContracts.RequestPermission`:

```
private final ActivityResultLauncher<String> requestPerm =
    registerForActivityResult(new ActivityResultContracts.RequestPermission(),
        new ActivityResultCallback<Boolean>() {
            @Override
            public void onActivityResult(Boolean wasGranted) {
                if (wasGranted) {
                    loadFromDir(Environment.getExternalStorageDirectory());
                }
                else {
                    Toast.makeText(MainActivity.this, R.string.msg_sorry, Toast.LENGTH_LONG).show();
                }
            }
        });
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
private val requestPerm = registerForActivityResult(ActivityResultContracts.RequestPermission()) {
    if (it) {
        loadFromDir(Environment.getExternalStorageDirectory())
    } else {
        Toast.makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG).show()
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

For the `openDoc` and `newDoc` items, we use `ActivityResultContracts.OpenDocument` and `ActivityResultContracts.CreateDocument`, respectively:

```
private final ActivityResultLauncher<String[]> openDoc =
    registerForActivityResult(new ActivityResultContracts.OpenDocument(),
        new ActivityResultCallback<Uri>() {
            @Override
            public void onActivityResult(Uri uri) {
                binding.text.setText("");
                motor.read(uri);
            }
        });

private final ActivityResultLauncher<String> createDoc =
    registerForActivityResult(new ActivityResultContracts.CreateDocument(),
        new ActivityResultCallback<Uri>() {
            @Override
            public void onActivityResult(Uri uri) {
                binding.text.setText("");
                motor.read(uri);
            }
        });
```

HANDLING FILES

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
private val openDoc =
    registerForActivityResult(ActivityResultContracts.OpenDocument()) { uri ->
        binding.text.setText("")
        uri?.let { motor.read(it) }
    }

private val createDoc =
    registerForActivityResult(ActivityResultContracts.CreateDocument()) { uri ->
        binding.text.setText("")
        uri?.let { motor.read(it) }
    }
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

In these cases, we get a Uri from the Storage Access Framework, so we do not need to convert anything into a Uri. And, in our case, we treat both the same: clear the EditText and pass the Uri to read():

In the end, no matter which of the options the user chooses, we pass a Uri to the MainMotor to read. Also, when the user clicks the save item, we pass that Uri and the current contents of the EditText to a write() function on the MainMotor.

Reading from the Location

Eventually, the read() call on MainMotor turns into a read() call on TextRepository.

Java

In Java, this creates and returns a LiveStreamReader implementation of LiveData that reads in the file and returns a StreamResult:

```
private static class LiveStreamReader extends LiveData<StreamResult> {
    private final Uri source;
    private final ContentResolver resolver;
    private final Executor executor;

    LiveStreamReader(Uri source, ContentResolver resolver, Executor executor) {
        this.source = source;
        this.resolver = resolver;
        this.executor = executor;

        postValue(new StreamResult(true, null, null, null));
    }
}
```

```
@Override
protected void onActive() {
    super.onActive();

    executor.execute(() -> {
        try {
            postValue(new StreamResult(false, source,
                slurp(resolver.openInputStream(source)), null));
        }
        catch (FileNotFoundException e) {
            postValue(new StreamResult(false, source, "", null));
        }
        catch (Throwable t) {
            postValue(new StreamResult(false, null, null, t));
        }
    });
}

private String slurp(final InputStream is) throws IOException {
    final char[] buffer = new char[8192];
    final StringBuilder out = new StringBuilder();
    final Reader in = new InputStreamReader(is, StandardCharsets.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);
    }

    is.close();

    return out.toString();
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.java](#))

StreamResult is just an encapsulation of the state, including the text once we have it loaded:

```
package com.commonsware.jetpack.contenteditor;

import android.net.Uri;

class StreamResult {
    final boolean isLoading;
    final Uri source;
    final String text;
}
```

HANDLING FILES

```
final Throwable error;

StreamResult(boolean isLoading, Uri source, String text, Throwable error) {
    this.isLoading = isLoading;
    this.source = source;
    this.text = text;
    this.error = error;
}
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/StreamResult.java](#))

Note that we use `ContentResolver` and `openInputStream()`. This not only works for a Storage Access Framework `Uri` but also one that we get from `Uri.fromFile()`. Most of `TextRepository` does not care what sort of `Uri` this is, so long as `ContentResolver` can open it.

Kotlin

The Kotlin version of `TextRepository` uses coroutines, returning a `StreamResult` directly from `read()`:

```
suspend fun read(context: Context, source: Uri) =
    withContext(Dispatchers.IO) {
        val resolver: ContentResolver = context.contentResolver

        try {
            resolver.openInputStream(source)?.use { stream ->
                StreamResult.Content(source, stream.reader().readText())
            } ?: throw IllegalStateException("could not open $source")
        } catch (e: FileNotFoundException) {
            StreamResult.Content(source, "")
        } catch (t: Throwable) {
            StreamResult.Error(t)
        }
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.kt](#))

Also, the Kotlin version of `StreamResult` is a sealed class representing the loading-content-error state:

```
package com.commonsware.jetpack.contenteditor

import android.net.Uri
```

```
sealed class StreamResult {
    object Loading : StreamResult()
    data class Content(val source: Uri, val text: String) : StreamResult()
    data class Error(val throwable: Throwable) : StreamResult()
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/StreamResult.kt](#))

Writing to the Location

Similarly, when we call `write()` on the motor, it routes to `write()` on the `TextRepository`.

Java

The Java version of `TextRepository` creates and returns a `LiveStreamWriter` that wraps up the disk I/O in a `LiveData`:

```
private static class LiveStreamWriter extends LiveData<StreamResult> {
    private final Uri source;
    private final ContentResolver resolver;
    private final Executor executor;
    private final String text;
    private Context context;

    LiveStreamWriter(Uri source, ContentResolver resolver, Executor executor,
                    String text, Context context) {
        this.source = source;
        this.resolver = resolver;
        this.executor = executor;
        this.text = text;
        this.context = context;

        postValue(new StreamResult(true, null, null, null));
    }

    @Override
    protected void onActive() {
        super.onActive();

        executor.execute(() -> {
            try {
                OutputStream os = resolver.openOutputStream(source);
                PrintWriter out = new PrintWriter(new OutputStreamWriter(os));

                out.print(text);
            }
        });
    }
}
```

```
        out.flush();
        out.close();

        final String externalRoot =
            Environment.getExternalStorageDirectory().getAbsolutePath();

        if (source.getScheme().equals("file") &&
            source.getPath().startsWith(externalRoot)) {
            MediaScannerConnection
                .scanFile(context,
                    new String[]{source.getPath()},
                    new String[]{"text/plain"},
                    null);
        }

        postValue(new StreamResult(false, source, text, null));
    }
    catch (Throwable t) {
        postValue(new StreamResult(false, null, null, t));
    }
    });
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.java](#))

We can just use `openOutputStream()` on `ContentResolver`, regardless of whether this is a Storage Access Framework `Uri` or one representing a file on the filesystem.

However, if the `Uri` *does* represent a file on the filesystem, and that file is on external storage, we want to tell the `MediaStore` about it. We detect this case by:

- Confirming that the scheme of the `Uri` is `file` — the scheme is the first portion of a URI, such as the `https` in `https://commonsware.com`
- Confirming that the path of the `Uri` starts with the root directory of external storage, to cover both our `external` and `externalRoot` scenarios

If both are true, we then call `scanFile()` on `MediaScannerConnection`, passing in:

- A `Context`
- An array of the paths to be indexed
- An array of the associated MIME types for each of those paths
- A callback object, or `null` if we do not need one (as is the case here)

Our file may not be indexed immediately, but it should be indexed much more

quickly than if we did not do this bit of work.

Kotlin

The Kotlin version of `TextRepository` does the same work, but once again it uses coroutines:

```
suspend fun write(context: Context, source: Uri, text: String) =
    withContext(Dispatchers.IO + appScope.coroutineContext) {
        try {
            val resolver = context.contentResolver

            resolver.openOutputStream(source)?.let { os ->
                PrintWriter(os.writer()).use { out ->
                    out.print(text)
                    out.flush()
                }
            }

            val externalRoot =
                Environment.getExternalStorageDirectory().absolutePath

            if (source.scheme == "file" &&
                source.path!!.startsWith(externalRoot)
            ) {
                MediaScannerConnection
                    .scanFile(
                        context,
                        arrayOf(source.path),
                        arrayOf("text/plain"),
                        null
                    )
            }

            StreamResult.Content(source, text)
        } catch (t: Throwable) {
            StreamResult.Error(t)
        }
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.kt](#))

In particular, we need to worry about the possibility that the user will use back navigation to leave the activity before our disk write is complete. When the activity is completely destroyed, its viewmodel is cleared, and that will cancel our coroutine. For a read operation, this is fine — we will not need that data anyway. But it would

be impolite to fail to write the data to disk (or, perhaps worse, write only part of it) just because the user left the activity. We need a separate `CoroutineScope`, one that will survive past the life of the viewmodel. For that, `TextRepository` has `appScope`:

```
private val appScope = CoroutineScope(SupervisorJob())
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.kt](#))

This `CoroutineScope` is configured with a `SupervisorJob`, which treats each job independently — if one job fails for some reason, other jobs will not be canceled automatically. We then use `appScope` (and its `CoroutineContext`) when launching our coroutine:

```
withContext(Dispatchers.IO + appScope.coroutineContext) {
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.kt](#))

This has the net effect of ensuring that our disk writes will not be interrupted by the viewmodel being cleared.

The Motor

Our motor's job is to work with `TextRepository` to load and save our text. As with most things, this is incrementally easier with Kotlin.

Java

Ideally, the activity would have a stable `LiveData` to observe for getting the content to display in the editor. However, our `TextRepository` returns a `LiveData` for each load request. Somehow, we need to “pour” all of those individual `LiveData` objects into a single `LiveData` that the activity observes.

The solution for that is `MediatorLiveData`.

`MediatorLiveData` can observe one or several other `LiveData` objects. When values change in those `LiveData` objects, `MediatorLiveData` will invoke a lambda expression that you provide. There, you can convert the value to whatever you need and update the `MediatorLiveData` itself based on that change. You can add and remove `LiveData` objects from the `MediatorLiveData` whenever you need.

So, `MainMotor` uses a `MediatorLiveData` as the stable `LiveData` that the activity observes. As we change sources of text content, we swap in a new `LiveData` source

for the MediatorLiveData:

```
package com.commonware.jetpack.contenteditor;

import android.app.Application;
import android.net.Uri;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;

public class MainMotor extends AndroidViewModel {
    private final TextRepository repo;
    private final MediatorLiveData<StreamResult> results = new MediatorLiveData<>();
    private LiveData<StreamResult> lastResult;

    public MainMotor(@NonNull Application application) {
        super(application);

        repo = TextRepository.get(application);
    }

    MediatorLiveData<StreamResult> getResults() {
        return results;
    }

    void read(Uri source) {
        if (lastResult != null) {
            results.removeSource(lastResult);
        }

        lastResult = repo.read(source);
        results.addSource(lastResult, results::postValue);
    }

    void write(Uri source, String text) {
        if (lastResult != null) {
            results.removeSource(lastResult);
        }

        lastResult = repo.write(source, text);
        results.addSource(lastResult, results::postValue);
    }
}
```

(from [ContentEditor/src/main/java/com/commonware/jetpack/contenteditor/MainMotor.java](#))

Note that we do not have a separate view-state class — instead, we just pass `StreamResult` along to the activity. That works in this case because we had no conversions that we needed to perform on the data from the repository.

Kotlin

The Kotlin version is simpler, using `MutableLiveData` and `viewModelScope` for getting the coroutine results over to `LiveData` and from there to `MainActivity`:

HANDLING FILES

```
package com.commonware.jetpack.contenteditor

import android.app.Application
import android.net.Uri
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _results = MutableLiveData<StreamResult>()
    val results: LiveData<StreamResult> = _results

    fun read(source: Uri) {
        _results.value = StreamResult.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = TextRepository.read(getApplication(), source)
        }
    }

    fun write(source: Uri, text: String) {
        _results.value = StreamResult.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = TextRepository.write(getApplication(), source, text)
        }
    }
}
```

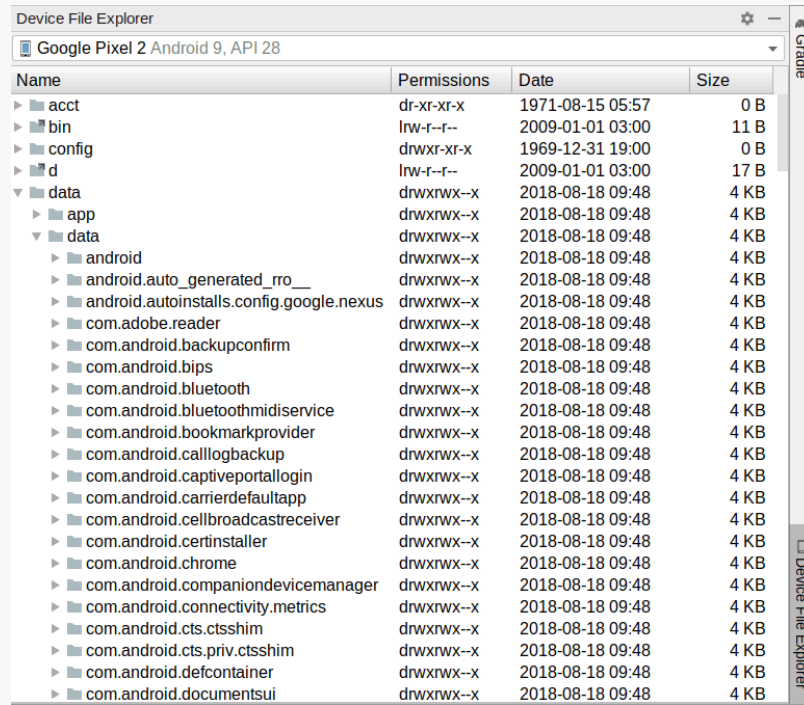
(from [ContentEditor/src/main/java/com/commonware/jetpack/contenteditor/MainMotor.kt](#))

The Results

The exact locations of internal, external, and removable storage may vary by device.

HANDLING FILES

On a typical device, each app's portion of internal storage can be found in `/data/data/...`, where the `...` is replaced by the application ID of the app. Unfortunately, this results in a very long list of apps, because all of the pre-installed apps get included:



Name	Permissions	Date	Size
acct	dr-xr-xr-x	1971-08-15 05:57	0 B
bin	lrw-r--r--	2009-01-01 03:00	11 B
config	drwxr-xr-x	1969-12-31 19:00	0 B
d	lrw-r--r--	2009-01-01 03:00	17 B
data	drwxrwx--x	2018-08-18 09:48	4 KB
app	drwxrwx--x	2018-08-18 09:48	4 KB
data	drwxrwx--x	2018-08-18 09:48	4 KB
android	drwxrwx--x	2018-08-18 09:48	4 KB
android.auto_generated_rro__	drwxrwx--x	2018-08-18 09:48	4 KB
android.autoinstalls.config.google.nexus	drwxrwx--x	2018-08-18 09:48	4 KB
com.adobe.reader	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.backupconfirm	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bips	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bluetooth	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bluetoothmidiservice	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bookmarkprovider	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.callogbackup	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.captiveportallogin	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.carrierdefaultapp	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.cellbroadcastreceiver	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.certinstaller	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.chrome	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.companiondevicemanager	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.connectivity.metrics	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.cts.ctsshim	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.cts.priv.ctsshim	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.defcontainer	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.documentsui	drwxrwx--x	2018-08-18 09:48	4 KB

Figure 220: Device File Explorer, Showing (Some) Internal Storage Locations

HANDLING FILES

If you type something into the app after you initially launch it, then click “Save”, then scroll to the `/data/data/com.commonsware.jetpack.contenteditor/` directory in the Device File Explorer, you should see the `files/` subdirectory that maps to `getFilesDir()`, and in there you should see a `test.txt` file that contains what you wrote:

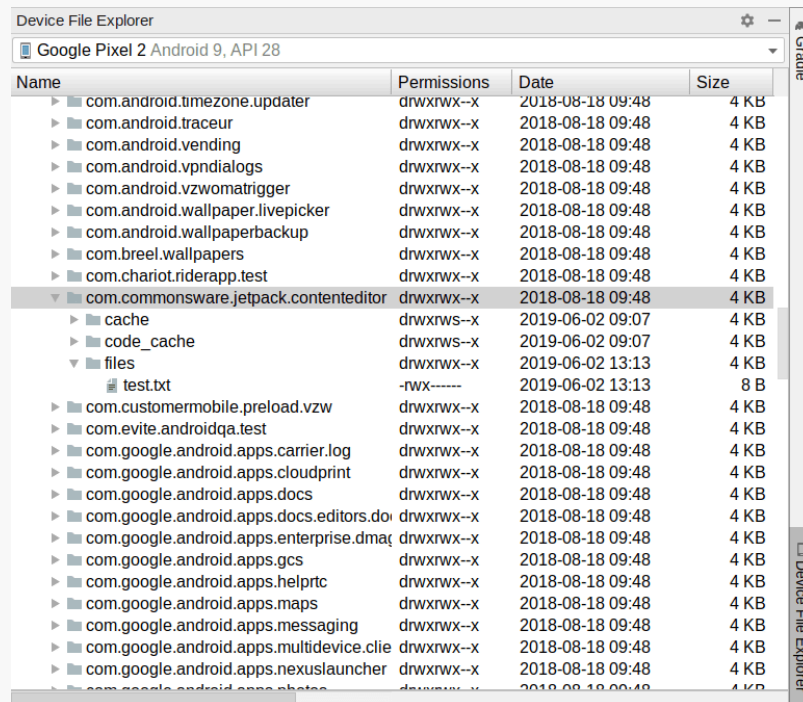


Figure 221: Device File Explorer, Showing Sample App’s Internal Storage Location

If you right-click over a file, a context menu gives you a few operations that you can perform on that file, such as “Save As” to download it to your development machine and “Delete” to remove it from the device. “Synchronize” updates the entire tree to reflect the current contents of the device.

HANDLING FILES

/sdcard in the Device File Explorer should give you a view of the root of external storage. This will include a test.txt file if you created it using the app by choosing “Load External Root” in the overflow menu, filling in some text, and clicking “Save”:

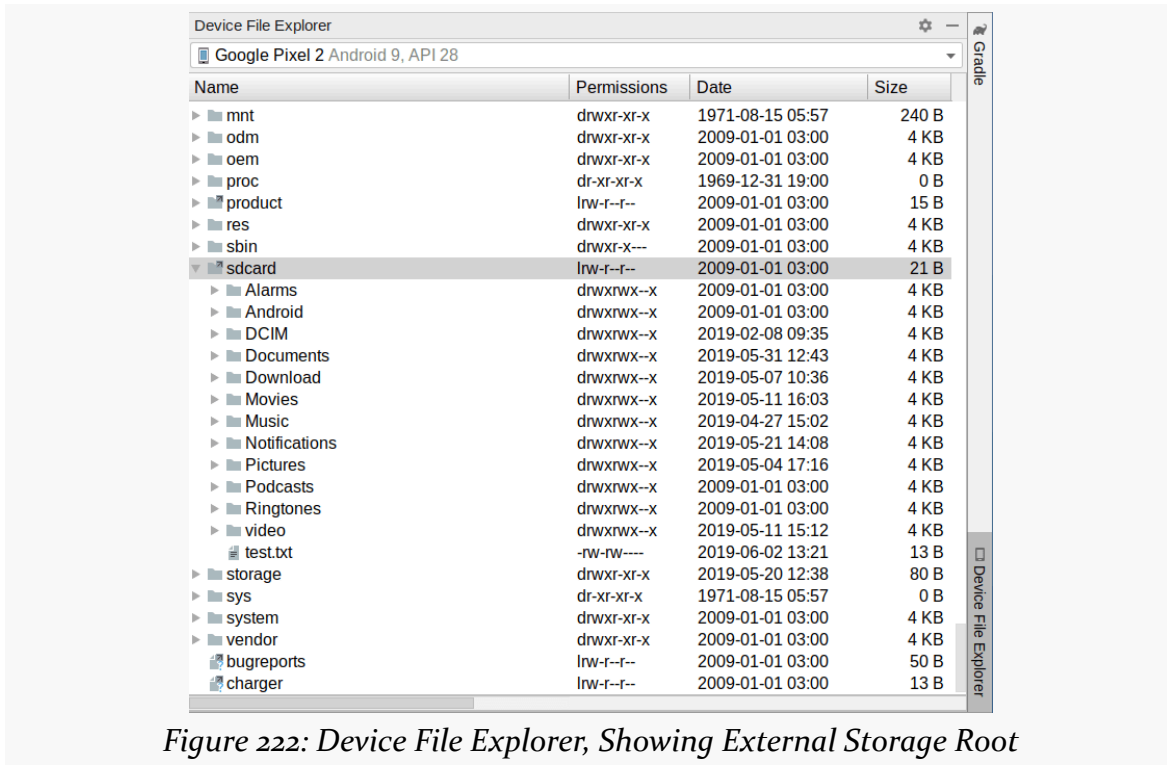


Figure 222: Device File Explorer, Showing External Storage Root

HANDLING FILES

There will be an `Android/` directory in the root of external storage, with a `data/` directory inside of it. That is akin to the `/data/data/` directory, except that it provides the list of app-specific external storage locations. And, fewer apps store data on external storage, so the list is more manageable, though it still can be rather long. If you use “Load External” in the app and save some text there, you will see a `test.txt` file in the app’s external storage location:

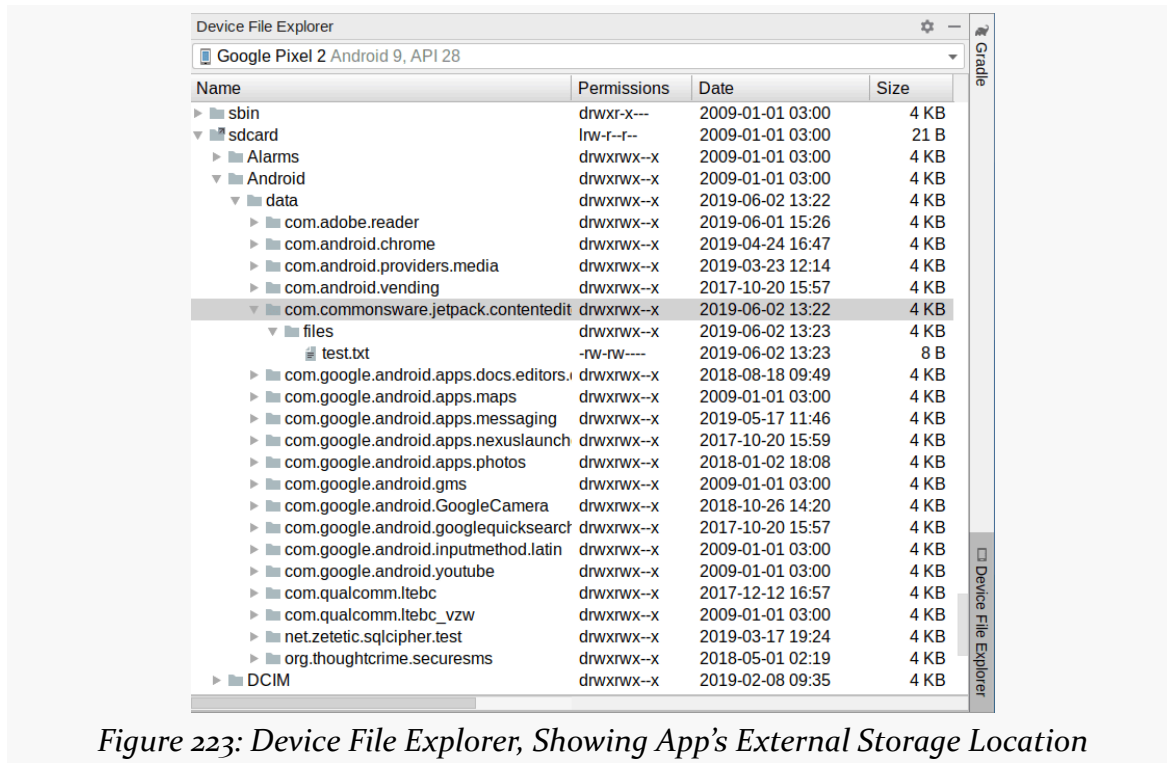


Figure 223: Device File Explorer, Showing App’s External Storage Location

Dealing with Android 10+

As noted earlier, methods like `Environment.getExternalStorageDirectory()` and their associated locations do not work on Android 10 and higher by default.

To deal with Android 10, in the manifest, we have

`android:requestLegacyExternalStorage="true"` on the `<application>` element:

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

  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

```
<application
    android:allowBackup="true"
    android:requestLegacyExternalStorage="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>
```

(from [ContentEditor/src/main/AndroidManifest.xml](#))

This gives Android 10 the same sorts of file access as Android 9.0 had.

Android 11+, though, will not completely work with just this attribute. Partly, that is because eventually we will need to raise our `targetSdkVersion` to 30, at which point `android:requestLegacyExternalStorage="true"` no longer works. But, more importantly, even *with* that attribute, we do not have write access to the root of external storage. We have write access to many other places, but not that one. As a result, we need to block access to the “Load External Root” item in our overflow menu on those devices.

To do that, we make use of version-specific resources.

In `res/values/`, each project has a `booleans.xml` file. By convention, this contains `<bool>` resources that define a boolean value:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="isPre11">true</bool>
</resources>
```

(from [ContentEditor/src/main/res/values/booleans.xml](#))

Here, we define `isPre11` to be true.

HANDLING FILES

Each project also has a `res/values-v30/` directory. This contains resources are only relevant on API Level 30+ devices, where API Level 30 corresponds to Android 11. In there, we have another `bools.xml` with its own definition of `isPre11`, setting the value to `false`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <bool name="isPre11">false</bool>
</resources>
```

(from [ContentEditor/src/main/res/values-v30/bools.xml](#))

Then, in `res/menu/actions.xml`, for the “Load External Root” item, we use `android:enabled` and point it to `@bool/isPre11`:

```
<item
  android:id="@+id/loadExternalRoot"
  android:title="@string/menu_external_root"
  android:enabled="@bool/isPre11"
  app:showAsAction="never" />
```

(from [ContentEditor/src/main/res/menu/actions.xml](#))

As the name suggests, `android:enabled` controls whether the item is enabled or not. This, plus our dual definitions of `isPre11`, means that:

- On Android 11+ devices, this item will be disabled, as Android will use the `res/values-v30/` edition of `isPre11`
- On older devices, this item will be enabled, as Android will fall back to the `res/values/` edition of `isPre11`

Our Java/Kotlin code can remain oblivious to this distinction, simply reacting to that item if it is chosen... even if on some devices, it cannot be chosen, because it is disabled.

Serving Files with FileProvider

We cannot assume that any other app has access to the files that we create, given Android’s increasing restrictions on file access. Our app has access to the files, and those on external or removable storage are accessible by the user. Beyond that, though, nothing is guaranteed.

If we want to allow another app to have access to our files, we need to use something

like `FileProvider`.

Scenarios for `FileProvider`

If we want to use `ACTION_VIEW` or various other Intent actions, we need to supply a `Uri` identifying what the action should be performed upon.

In early versions of Android, you might have written the file to external storage, then used `Uri.fromFile()` to get a `file://Uri` that points to the file that you created. However, that has a few problems:

- It has been years since you could assume that every app requested `READ_EXTERNAL_STORAGE`
- You are cluttering up the user's external storage with files that the user may or may not want there
- On Android 7.0, `file://Uri` values were mostly blocked by the OS, if you try to pass one in an Intent

If you used the Storage Access Framework and wrote your content to a location specified by one of its `Uri` values, you can use that `Uri` with `ACTION_VIEW`. However, this implies that the user has a use for this content independent of both apps (yours and whatever responds to the `ACTION_VIEW` Intent). That may not be the case. For example, your app might package a PDF file to serve as a user manual. Your app has the PDF, and a PDF viewer needs access to it, but the user does not necessarily need (or even want) that PDF to be stored somewhere public.

Those are the scenarios where `FileProvider` is useful: for content that your app has, that other apps need, but the user does not need independently of those apps.

Configuring `FileProvider`

The `PdfProvider` sample module in the [Sampler](#) and [SamplerJ](#) projects implements the scenario outlined above:

- We have a PDF packaged with the app as an asset
- We want to allow a PDF viewer to view that PDF, so the user can read it

One way to handle this is to copy the asset to a file, then use `FileProvider` to make it available to the PDF viewer.

Metadata XML Resource

We need to teach `FileProvider` what files we are willing to provide to other apps. To do that, we need to define an xml resource, in the same `res/xml/` directory where we put our preference screen configuration. `res/xml/` is a resource directory that can hold any XML that you want, including arbitrary XML that you create. Some parts of Android, like the preference system and `FileProvider`, will want a resource matching their desired XML structure.

In the case of `FileProvider`, that consists of a root `<paths>` element, followed by one or more child elements indicating where we want to serve from and what we want that location to be called:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
  <files-path name="stuff" path="/" />
</paths>
```

(from [PdfProvider/src/main/res/xml/provider_paths.xml](#))

`<files-path>` says “serve from `getFilesDir()` as a root directory”. That is one of a few possible element names, including `<cache-path>` (mapping to `getCacheDir()`) and `<external-files-path>` (mapping to `getExternalFilesDir(null)`).

The `path` attribute indicates where underneath the specified root location we want to serve. Here, by using `/`, we are saying that anything in `getFilesDir()` is fine. If there are files in `getFilesDir()` that you do not want to be available via `FileProvider`, set up a designated directory under `getFilesDir()` for the shareable files, then put that directory name in the `path` attribute.

The `name` attribute is a unique name for this location. No two locations in your resource can share the same name. This will form part of the `Uri` that `FileProvider` uses to map to your files.

Manifest Element

`FileProvider` is an implementation of `ContentProvider`. A `ContentProvider`, like an `Activity`, needs to be registered in the manifest. Instead of an `<activity>` element, it uses a `<provider>` element, but otherwise it fills the same basic role: tell Android that this class is an entry point for our app and how it can be used.

So, our manifest has a `<provider>` element, pointing to the `AndroidX`

implementation of FileProvider:

```
<provider
  android:name="androidx.core.content.FileProvider"
  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 [PdfProvider/src/main/AndroidManifest.xml](#))

As with `<activity>`, the `android:name` attribute on `<provider>` points to the class that is the `ContentProvider` implementation. Since this one is coming from a library, we need to provide the fully-qualified class name (`androidx.core.content.FileProvider`).

`android:authorities` is an identifier (or optionally several in a comma-delimited list). This identifier has to be unique on the device; there cannot be two `<provider>` elements with the same authority installed at the same time. Fortunately, the Android build system lets us use the `${applicationId}` macro, which expands into our application ID. This app's application ID is `com.commonware.jetpack.pdfprovider`, so our authority turns into `com.commonware.jetpack.pdfprovider.provider`.

`android:exported` says whether or not third-party apps can initiate communications on their own with this `ContentProvider`. `FileProvider` requires this to be set to `false`. The only way another app will be able to work with our content is if we grant it temporary permission, on a case-by-case basis.

`android:grantUriPermissions` says whether or not we want to use that “case-by-case basis” approach or not. `true` indicates that we do.

Finally, as we saw in [the chapter on app widgets](#), the nested `<meta-data>` element is a way that you can add configuration details to the manifest. In this case, `FileProvider` expects a `<meta-data>` element for `android.support.FILE_PROVIDER_PATHS`, pointing to the `xml` resource that we saw in the preceding section.

With that resource and this manifest element, our `FileProvider` is ready for use.

Employing FileProvider

If you have a file in one of the locations listed in your <paths> resource, you can call `FileProvider.getUriForFile()` to retrieve the corresponding Uri:

```
binding.view.setOnClickListener(v -> {
    Uri uri = FileProvider.getUriForFile(this, AUTHORITY, state.content);
    Intent intent =
        new Intent(Intent.ACTION_VIEW)
            .setDataAndType(uri, "application/pdf")
            .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

    try {
        startActivity(intent);
    }
    catch (ActivityNotFoundException ex) {
        Toast.makeText(this, "Sorry, we cannot display that PDF!",
            Toast.LENGTH_LONG).show();
    }
});
```

(from PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.java)

```
binding.view.setOnClickListener {
    val uri = FileProvider.getUriForFile(this, AUTHORITY, state.pdf)
    val intent = Intent(Intent.ACTION_VIEW)
        .setDataAndType(uri, "application/pdf")
        .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)

    try {
        startActivity(intent);
    } catch (ex: ActivityNotFoundException) {
        Toast.makeText(
            this,
            "Sorry, we cannot display that PDF!",
            Toast.LENGTH_LONG
        ).show()
    }
}
```

(from PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.kt)

Here, `state.pdf` is a `File` object, pointing to a PDF file that we have copied from assets to a file via some code in `MainMotor`. This is `MainActivity`. `AUTHORITY` is the authority that we used when declaring the `FileProvider` in the manifest:

HANDLING FILES

```
private static final String AUTHORITY =  
    BuildConfig.APPLICATION_ID + ".provider";
```

(from PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.java)

```
private const val AUTHORITY = "${BuildConfig.APPLICATION_ID}.provider"
```

(from PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.kt)

The equivalent of the `${applicationId}` macro in the manifest is to refer to `BuildConfig.APPLICATION_ID`, so we are assembling the authority string the same way.

Given the `Uri` from `FileProvider`, we can put it in an `ACTION_VIEW` Intent and pass that to `startActivity()`, to try to view the PDF file. However, by default, third-party apps have no rights to view the content identified by that `Uri`. By calling `addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)` on the Intent, we are telling Android to grant read access (but not write access) to that content.

Overall, the activity has a pair of buttons, one to copy the PDF from the asset to a file in `getFilesDir()`:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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=".MainActivity">  
  
    <Button  
        android:id="@+id/export"  
        android:layout_width="0dp"  
        android:layout_height="wrap_content"  
        android:layout_marginEnd="8dp"  
        android:layout_marginStart="8dp"  
        android:layout_marginTop="8dp"  
        android:text="@string/btn_export"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintStart_toStartOf="parent"  
        app:layout_constraintTop_toTopOf="parent" />  
  
    <Button  
        android:id="@+id/view"  
        android:layout_width="0dp"
```

HANDLING FILES

```
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:enabled="false"
        android:text="@string/btn_view"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/export" />

<TextView
    android:id="@+id/error"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:typeface="monospace"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/view" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from PdfProvider/src/main/res/layout/activity_main.xml)

The work to copy the asset to a file is handled by an `exportPdf()` function on `MainMotor`:

```
package com.commonware.jetpack.pdfprovider;

import android.app.Application;
import android.content.res.AssetManager;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;

public class MainMotor extends AndroidViewModel {
    private static final String FILENAME = "test.pdf";
    private final MutableLiveData<MainViewState> states = new MutableLiveData<>();
    private final AssetManager assets;
    private final File dest;
    private final Executor executor = Executors.newSingleThreadExecutor();
```

HANDLING FILES

```
public MainMotor(@NonNull Application application) {
    super(application);

    assets = application.getAssets();
    dest = new File(application.getFilesDir(), FILENAME);

    if (dest.exists()) {
        states.setValue(new MainViewState(false, dest, null));
    }
}

LiveData<MainViewState> getStates() {
    return states;
}

void exportPdf() {
    states.setValue(new MainViewState(true, null, null));

    executor.execute(() -> {
        try {
            copy(assets.open(FILENAME));
            states.postValue(new MainViewState(false, dest, null));
        }
        catch (IOException e) {
            states.postValue(new MainViewState(false, null, e));
        }
    });
}

private void copy(InputStream in) throws IOException {
    FileOutputStream out=new FileOutputStream(dest);
    byte[] buf=new byte[8192];
    int len;

    while ((len=in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }

    in.close();
    out.getFD().sync();
    out.close();
}
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainMotor.java](#))

```
package com.commonsware.jetpack.pdfprovider

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import java.io.File
import java.io.FileOutputStream
```

HANDLING FILES

```
import java.io.IOException

private const val FILENAME = "test.pdf"

class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _states = MutableLiveData<MainViewState>()
    val states: LiveData<MainViewState> = _states
    private val assets = application.assets
    private val dest = File(application.filesDir, FILENAME)

    init {
        if (dest.exists()) {
            _states.value = MainViewState.Content(dest)
        }
    }

    fun exportPdf() {
        _states.value = MainViewState.Loading

        viewModelScope.launch(Dispatchers.IO) {
            try {
                assets.open(FILENAME).use { pdf ->
                    FileOutputStream(dest).use { pdf.copyTo(it) }
                }
                _states.postValue(MainViewState.Content(dest))
            } catch (e: IOException) {
                _states.postValue(MainViewState.Error(e))
            }
        }
    }
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainMotor.kt](#))

MainMotor does that work on a background thread, since it might take a few moments. It emits MainViewState objects to report our loading/content/error status:

```
package com.commonsware.jetpack.pdfprovider;

import java.io.File;

class MainViewState {
    final boolean isLoading;
    final File content;
    final Throwable error;
```

HANDLING FILES

```
MainViewState(boolean isLoading, File content, Throwable error) {
    this.isLoading = isLoading;
    this.content = content;
    this.error = error;
}
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainViewState.java](#))

```
package com.commonsware.jetpack.pdfprovider

import java.io.File

sealed class MainViewState {
    object Loading : MainViewState()
    class Content(val pdf: File) : MainViewState()
    class Error(val throwable: Throwable) : MainViewState()
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainViewState.kt](#))

MainActivity observes the LiveData of MainViewState, and when the content is ready, enables the second button. When that button is clicked, we start the PDF viewer activity to view our content.

```
package com.commonsware.jetpack.pdfprovider;

import android.content.ActivityNotFoundException;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;
import com.commonsware.jetpack.pdfprovider.databinding.ActivityMainBinding;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.content.FileProvider;
import androidx.lifecycle.ViewModelProvider;

public class MainActivity extends AppCompatActivity {
    private static final String AUTHORITY =
        BuildConfig.APPLICATION_ID + ".provider";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final ActivityMainBinding binding =
```

HANDLING FILES

```
        ActivityMainBinding.inflate(getLayoutInflater());

        setContentView(binding.getRoot());

        final MainMotor motor = new ViewModelProvider(this).get(MainMotor.class);

        motor.getStates().observe(this, state -> {
            binding.export.setEnabled(!state.isLoading && state.content == null);
            binding.view.setEnabled(!state.isLoading && state.content != null);

            if (binding.view.isEnabled()) {
                binding.view.setOnClickListener(v -> {
                    Uri uri = FileProvider.getUriForFile(this, AUTHORITY, state.content);
                    Intent intent =
                        new Intent(Intent.ACTION_VIEW)
                            .setDataAndType(uri, "application/pdf")
                            .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

                    try {
                        startActivity(intent);
                    }
                    catch (ActivityNotFoundException ex) {
                        Toast.makeText(this, "Sorry, we cannot display that PDF!",
                            Toast.LENGTH_LONG).show();
                    }
                });
            }

            if (state.error != null) {
                binding.error.setText(state.error.getLocalizedMessage());
            }
        });

        binding.export.setOnClickListener(v -> motor.exportPdf());
    }
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.java](#))

```
package com.commonsware.jetpack.pdfprovider

import android.content.ActivityNotFoundException
import android.content.Intent
import android.os.Bundle
import android.widget.Toast
import androidx.activity.viewModels
import androidx.appcompat.app.AppCompatActivity
import androidx.core.content.FileProvider
```

```
import com.commonware.jetpack.pdfprovider.databinding.ActivityMainBinding

private const val AUTHORITY = "${BuildConfig.APPLICATION_ID}.provider"

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)

        val motor: MainMotor by viewModels()

        motor.states.observe(this) { state ->
            when (state) {
                MainViewState.Loading -> {
                    binding.export.isEnabled = false
                    binding.view.isEnabled = false
                }
                is MainViewState.Content -> {
                    binding.export.isEnabled = false
                    binding.view.isEnabled = true
                    binding.view.setOnClickListener {
                        val uri = FileProvider.getUriForFile(this, AUTHORITY, state.pdf)
                        val intent = Intent(Intent.ACTION_VIEW)
                            .setDataAndType(uri, "application/pdf")
                            .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)

                        try {
                            startActivity(intent);
                        } catch (ex: ActivityNotFoundException) {
                            Toast.makeText(
                                this,
                                "Sorry, we cannot display that PDF!",
                                Toast.LENGTH_LONG
                            ).show()
                        }
                    }
                }
                is MainViewState.Error -> {
                    binding.export.isEnabled = false
                    binding.view.isEnabled = false
                    binding.error.text = state.throwable.localizedMessage
                }
            }
        }
    }
}
```



```
    }  
  
    binding.export.setOnClickListener { motor.exportPdf() }  
  }  
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.kt](#))

What You Should Use

For data that your app needs, but that the user only needs through your app, use internal storage. This can include data that the user might consume with some other app, such as in the PDF FileProvider. With internal storage, your app needs to initiate giving the user the data, whether it is directly in your UI or by serving it up to other apps.

For data that the user will want independently of your app, use the Storage Access Framework, due to the long-term limitations of working with external and removable storage via the filesystem. External or removable storage should be limited to cases where you are using some third-party library that only accepts a `File` as input, or similar cases where you are unable to use a `Uri` from the Storage Access Framework.

Accessing the Internet

Nearly every Android device has Internet access. It is fairly likely that you will have interest in accessing the Internet from your Android app.

Android has a lot to offer here, both in terms of what is part of the OS and in terms of the wide range of add-on libraries to assist in Internet access.

This chapter covers some general Internet access options for Android apps, then walks through a sample using two popular libraries: Retrofit and Glide.

NOTE: This chapter will use “the Internet” to refer to any networking. The rules and options outlined here are not solely for accessing the public Internet but also for accessing private networks, such as an office network.

An API Roundup

There are lots and lots of ways that your app can access the Internet. Some APIs are part of the Android SDK, while some are third-party libraries layered atop the SDK. Here are some of the more popular and better-known options.

Socket

In the end, anything in Java/Kotlin code that is working with the Internet will wind up using a Socket. If you are trying to create a library that implements some obscure or new Internet protocol, you might find yourself using Socket to connect to some server, then implementing all of the communications yourself.

Most developers should not need to do this. Most major protocols have implementations available for Android, whether in the OS or in a library. All else

being equal, you are better served using an existing implementation rather than “rolling your own”.

URLConnection

The original HTTP API in Java centers around `URLConnection` and related classes. Android supports this, and it is the only supported direct HTTP API in the Android SDK at the present time. However, the `URLConnection` API is *very* old, and there are better alternatives, such as `OkHttp`. In fact, Android’s `URLConnection` is built on top of a forked copy of `OkHttp`.

WebView

If your objective is to display Web content to the user, most likely you will want to use `WebView`. This is a widget in the Android SDK that renders Web content, much like a browser. You can put a `WebView` in your activities or fragments and hand the `WebView` the content to display, in the form of HTML strings or URLs.

Conversely, if you are looking to talk to a Web service or otherwise engage in HTTPS communications without displaying HTML/CSS/JavaScript to the user, you will want to use something other than `WebView`.

DownloadManager

If your objective is to download a file from a publicly-accessible URL, you could use `DownloadManager`. This too is part of the Android SDK, and its job is to perform this sort of download. It handles some of the complexity for you, such as dealing with network disconnections and picking up the download later from where it left off.

However, `DownloadManager` has many limitations. For example, you have no means of providing a session cookie or authentication header to the Web server — the URL must be one that can be used without additional HTTP stuff. You have limited places for storing the downloaded result. By default, the user will see a notification in their status bar, indicating the progress of the download. And so on.

For its narrow purpose, `DownloadManager` may be OK, but is not designed for working with arbitrary Web services and does not offer much flexibility.

Volley

Google offers its own add-on library for Internet access, called [Volley](#). Volley offers a cleaner API for accessing Web services and downloading images than you get with `URLConnection`, even though “under the covers” Volley uses `URLConnection` to do its work. Beyond that, Volley’s primary “claims to fame” are:

- It was created by Google
- It reportedly is used by the Play Store and some other Google apps

As such, some developers prefer to use this, over other options, as it is easier to get the library approved by decision-makers.

OkHttp

Perhaps the most popular option for generic HTTP requests is [OkHttp](#). Published by Square, OkHttp is frequently updated and has good developer and community support. OkHttp is a first-class HTTP client, working directly with `Socket` (as opposed to being layered atop `URLConnection`). OkHttp has a clean yet rich API for making and monitoring requests and responses.



You can learn more about OkHttp in the "Contacting a Web Service" chapter of [Exploring Android](#)!

OkHttp also serves as the “plumbing” for various other libraries, including most of those listed below.

Retrofit

Square’s [Retrofit](#) is a library layered atop of OkHttp that aims to simplify making REST-style Web service calls. You use annotations on a Java/Kotlin interface to describe the REST URLs that you wish to access, along with the HTTP operations to perform (e.g., GET, POST, PUT). If you combine Retrofit with a parsing library suitable for the Web service — such as a JSON parser for Web services serving JSON responses — Retrofit allows you to craft an API that feels like it is just a local function call yet returns an object tree representing the response.

We will examine the use of Retrofit in [the sample app](#).

Apollo-Android

If your server is using GraphQL instead of REST, [Apollo-Android](#) is basically “Retrofit for GraphQL”. You provide your GraphQL in the form of documents in a `graphql/` project directory. Apollo-Android plugins will generate Java/Kotlin classes that implement that API and represent the responses. You can then use that API at runtime, with OkHttp handling the HTTPS communications.

Image Loaders

A common need in Android apps is to display images from URLs. There are *lots* of image-loading libraries that handle that work for you with a simple API. The two most popular are [Glide](#) and [Picasso](#). Both can integrate with OkHttp, which is particularly important if you are using OkHttp-based libraries elsewhere and wish to have a common configuration for things like logging, proxy servers, and so on.

We will examine the use of Glide in [the sample app](#).

Specialized APIs

If you are looking to communicate with an existing third-party Web service, there might be dedicated libraries for that purpose. That third party might offer their own SDK, and independent developers may have created other libraries.

In general, if a Web service has an official SDK, you are better off using that. If nothing else, when it comes time to get support, the Web service developers may be expecting you to use their SDK and may have difficulty helping you debug your own manual HTTPS code.

Android’s Restrictions

To a large extent, accessing the Internet on Android is not significantly different than is accessing the Internet on other platforms. For example, Apache HttpClient, OkHttp, and Retrofit work with ordinary Java/Kotlin for use on desktops and servers, in addition to working on Android. Similarly, WebView is backed by the same Web rendering engine that powers Chrome, not only for Android, but for all Chrome-supported platforms.

However, Android *does* have some specific restrictions that will impact how you work with the Internet. None should be big problems for you, but they will require

some extra care to handle.

The INTERNET 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"/>
```

Note, though, that you do not need to request this permission at runtime — just having the `<uses-permission>` element in the manifest is sufficient.

NetworkOnMainThreadException

If you attempt to perform network I/O on the main application thread, you will crash immediately with a `NetworkOnMainThreadException`. Network I/O can be slow — even if it is fast for you in the office, it may be slow for users due to differences in network connectivity. As such, doing network I/O on the main application thread is very likely to cause your UI to freeze for a bit while that I/O goes on.

Technically, there are ways to disable the code that enforces this exception (a class called `StrictMode`). In practice, the right answer is to do your network I/O on a background thread.

Many of the HTTP options presented above have built-in thread management options:

- `WebView`
- `DownloadManager`
- `Volley`
- `OkHttp`
- `Retrofit`
- `Apollo-Android`
- `Glide`
- `Picasso`

In some cases, everything is asynchronous (e.g., `WebView`, `DownloadManager`). In

other cases, you have your choice between synchronous and asynchronous APIs (e.g., `OkHttp`, `Retrofit`) — you would use the synchronous APIs in cases where you are doing something else to put the work in the background, such as Kotlin coroutines.

Cleartext Restrictions

On Android 8.0+, if you attempt to use an `http` URL, you will crash, as “cleartext” traffic is not allowed by default.

The best answer is to use an `https` URL. In particular, if you are working with a server that you control, please secure it with an SSL/TLS certificate and use `https`.

If you are not in control over the schemes used in the URLs, though, you can update your [network security configuration](#) to support cleartext traffic from some or all domains. Consider warning your users, though, when you are about to use an `http` URL.

The Reality of Mobile Devices

Mobile devices have an interesting property: they are mobile. Users with mobile devices have this annoying habit of walking, jogging, running, driving, riding, dancing, kayaking, skydiving, and so on. When they do that, their network connection may change:

- They switch from their home or office WiFi to a mobile data connection
- They switch from one mobile tower to another tower while they are “on the road” (or in the sky, on the water, etc.)
- They return to their home or office and reconnect to the WiFi
- They get in an elevator, or go into an underground parking garage, or otherwise move from an area with connectivity options to an area without them

In all these cases, when the network connection changes, any outstanding socket connections get dropped. It does not matter whether you are working directly with `Socket` or are working with HTTP APIs. So, you might be in the middle of getting a Web service reply and, all of a sudden, your request gets abandoned, because you (briefly) lost communication with the server.

In casual apps, like this book’s samples, you largely ignore this issue. In production-grade apps, typically you handle this by having suitable retry policies. So, if you get

an `IOException`, you retry the request up to *N* times before giving up. That way, for a transient problem like a connectivity change, you get the work done, just a bit more slowly. Yet, at the same time, you do not retry indefinitely, as if there are lots of successive failures, it may require user intervention to fix (e.g., the user turned on airplane mode).

Forecasting the Weather

The Weather sample module in the [Sampler](#) and [SamplerJ](#) projects shows a weather forecast for New York City. It gets the forecast from the US National Weather Service, which offers a Web service for retrieving forecasts. The forecast elements that the app uses includes the time, projected temperature, and an icon representing the expected weather (e.g., sunny, partly cloudy, mostly cloudy, rain, snow, fog, alien invasion, kaiju attack, or zombie infestation).

(OK, perhaps not those last three)

The icon is supplied in the form of a URL from which we can download the icon.

To get the weather forecast itself, we will use Retrofit. To populate `ImageView` widgets in a `RecyclerView` with the weather icons, we will use Glide.

The Dependencies

Glide is simple. Most of the time, you can just use the `com.github.bumptech.glide:glide` library:

```
implementation 'com.github.bumptech.glide:glide:4.12.0'
```

(from [Weather/build.gradle](#))

Retrofit is a bit more complicated. You will usually need at least two libraries:

- `com.squareup.retrofit2:retrofit`, which is Retrofit itself
- A “converter” library that teaches Retrofit how to parse the sort of content that your Web service returns (JSON, XML, etc.)

The US National Weather Service Web service API supports returning JSON. There are a few JSON converter options for Retrofit, each of which use a different JSON parser. If you are using a JSON parser elsewhere in your app, you can try to use the corresponding Retrofit converter library — that way, you do not wind up bundling

two JSON parsing libraries in your app. The three most popular JSON parsers for Android are Gson, Jackson, and Moshi, and there are Retrofit converters for each of those. [Moshi](#) happens to be written by Square, the same team that created Retrofit itself, so this sample app uses Moshi:

```
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-moshi:2.9.0"
```

(from [Weather/build.gradle](#))

The Response Classes

The converters that Retrofit supports are designed to take a raw response — such as a JSON string — and convert them into instances of classes that are designed to reflect the response structure. The US National Weather Service has [some documentation](#) about their Web service API, and that information can be used to create classes that match the JSON that they will serve.

Usually, your classes can skip portions of the response that you do not care about. Moshi or other JSON parsers will skip anything that appears in the JSON but does not have a corresponding spot in your Java or Kotlin classes.

The JSON that we get back from the Web service looks like this (with the list of forecast periods truncated to save space in the book):

```
{
  "@context": [
    "https://raw.githubusercontent.com/geojson/geojson-ld/master/contexts/geojson-
base.jsonld",
    {
      "wx": "https://api.weather.gov/ontology#",
      "geo": "https://www.opengis.net/ont/geosparql#",
      "unit": "https://codes.wmo.int/common/unit/",
      "@vocab": "https://api.weather.gov/ontology#"
    }
  ],
  "type": "Feature",
  "geometry": {
    "type": "GeometryCollection",
    "geometries": [
      {
        "type": "Point",
        "coordinates": [
          -74.0130202,
          40.714515800000001
        ]
      }
    ]
  }
}
```

```
    ]
  },
  {
    "type": "Polygon",
    "coordinates": [
      [
        [
          -74.025095199999996,
          40.727052399999998
        ],
        [
          -74.0295579,
          40.705361699999997
        ],
        [
          -74.0009483000000005,
          40.701977499999998
        ],
        [
          -73.9964798000000003,
          40.7236678000000001
        ],
        [
          -74.025095199999996,
          40.727052399999998
        ]
      ]
    ]
  }
],
"properties": {
  "updated": "2019-06-05T19:48:39+00:00",
  "units": "us",
  "forecastGenerator": "BaselineForecastGenerator",
  "generatedAt": "2019-06-05T22:38:08+00:00",
  "updateTime": "2019-06-05T19:48:39+00:00",
  "validTimes": "2019-06-05T13:00:00+00:00/P8D",
  "elevation": {
    "value": 2.1335999999999999,
    "unitCode": "unit:m"
  },
  "periods": [
    {
      "number": 1,
      "name": "Tonight",
      "startTime": "2019-06-05T18:00:00-04:00",
      "endTime": "2019-06-06T06:00:00-04:00",
```

ACCESSING THE INTERNET

```
        "isDaytime": false,
        "temperature": 67,
        "temperatureUnit": "F",
        "temperatureTrend": null,
        "windSpeed": "10 to 14 mph",
        "windDirection": "SW",
        "icon": "https://api.weather.gov/icons/land/night/
tsra,80?size=medium",
        "shortForecast": "Showers And Thunderstorms",
        "detailedForecast": "Showers and thunderstorms. Cloudy, with a low
around 67. Southwest wind 10 to 14 mph, with gusts as high as 24 mph. Chance of
precipitation is 80%. New rainfall amounts between a quarter and half of an inch
possible."
    },
    {
        "number": 2,
        "name": "Thursday",
        "startTime": "2019-06-06T06:00:00-04:00",
        "endTime": "2019-06-06T18:00:00-04:00",
        "isDaytime": true,
        "temperature": 83,
        "temperatureUnit": "F",
        "temperatureTrend": null,
        "windSpeed": "10 mph",
        "windDirection": "W",
        "icon": "https://api.weather.gov/icons/land/day/rain_showers,50/
rain_showers,20?size=medium",
        "shortForecast": "Chance Rain Showers",
        "detailedForecast": "A chance of rain showers before 3pm. Partly
sunny, with a high near 83. West wind around 10 mph. Chance of precipitation is 50%.
New rainfall amounts between a tenth and quarter of an inch possible."
    },
    {
        "number": 3,
        "name": "Thursday Night",
        "startTime": "2019-06-06T18:00:00-04:00",
        "endTime": "2019-06-07T06:00:00-04:00",
        "isDaytime": false,
        "temperature": 66,
        "temperatureUnit": "F",
        "temperatureTrend": null,
        "windSpeed": "7 to 10 mph",
        "windDirection": "N",
        "icon": "https://api.weather.gov/icons/land/night/sct?size=medium",
        "shortForecast": "Partly Cloudy",
        "detailedForecast": "Partly cloudy, with a low around 66. North wind
7 to 10 mph."
    }
}
```

```
    ]  
  }  
}
```

For the purposes of this app, we only need a small subset of this information. We need the periods list of objects from the properties property of the root JSON object. And, for each period, we need the name, temperature, temperatureUnit, and icon properties. A full-featured US weather app could use a lot more of the properties, but we will keep this simple.

So, we have these classes to model that response in Java:

```
package com.commonware.jetpack.weather;  
  
import java.util.List;  
  
public class WeatherResponse {  
    public final Properties properties=null;  
  
    public static class Properties {  
        public final List<Forecast> periods=null;  
    }  
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/WeatherResponse.java](#))

```
package com.commonware.jetpack.weather;  
  
public class Forecast {  
    final String name;  
    final int temperature;  
    final String temperatureUnit;  
    final String icon;  
  
    public Forecast(String name, int temperature,  
                    String temperatureUnit, String icon) {  
        this.name = name;  
        this.temperature = temperature;  
        this.temperatureUnit = temperatureUnit;  
        this.icon = icon;  
    }  
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/Forecast.java](#))

...and Kotlin:

```
package com.commonware.jetpack.weather

class WeatherResponse {
    val properties: Properties? = null

    class Properties {
        val periods: List<Forecast>? = null
    }
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/WeatherResponse.kt](#))

```
package com.commonware.jetpack.weather

data class Forecast(
    val name: String,
    val temperature: Int,
    val temperatureUnit: String,
    val icon: String
)
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/Forecast.kt](#))

The overall Web service response is a `WeatherResponse`, which holds a `Properties` object, which in turn host a `List` of `Forecast` objects.

Moshi does not care about the package structure, so long as it can create instances of the class and fill in the fields or properties. Here, we have a mix of top-level classes (`WeatherResponse`, `Forecast`) and nested classes (`Properties`) — you can organize your classes as you see fit.

The Retrofit API Declaration

The next step for using Retrofit is to declare an interface that describes the API that we are invoking on the Web service and maps it to functions that we want to be able to call from our Java/Kotlin code.

In our case, we are only invoking a single Web service URL, where we tell it a location for a forecast, and we get back the corresponding forecast data. So, our interface has only a `getForecast()` function:

```
package com.commonware.jetpack.weather;

import retrofit2.Call;
import retrofit2.http.GET;
```

ACCESSING THE INTERNET

```
import retrofit2.http.Headers;
import retrofit2.http.Path;

public interface NWSInterface {
    @Headers("Accept: application/geo+json")
    @GET("/gridpoints/{office}/{gridX},{gridY}/forecast")
    Call<WeatherResponse> getForecast(@Path("office") String office,
                                     @Path("gridX") int gridX,
                                     @Path("gridY") int gridY);
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/NWSInterface.java](#))

```
package com.commonsware.jetpack.weather

import retrofit2.http.GET
import retrofit2.http.Headers
import retrofit2.http.Path

interface NWSInterface {
    @Headers("Accept: application/geo+json")
    @GET("/gridpoints/{office}/{gridX},{gridY}/forecast")
    suspend fun getForecast(
        @Path("office") office: String,
        @Path("gridX") gridX: Int,
        @Path("gridY") gridY: Int
    ): WeatherResponse
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/NWSInterface.kt](#))

Every Retrofit interface function will have at least one annotation, one that indicates the HTTP operation to perform and a relative path on which to perform it. In our case, that is `@GET("/gridpoints/{office}/{gridX},{gridY}/forecast")`, saying that we want to perform an HTTP GET operation on... something.

The `@GET` annotation takes a relative path, where portions of that path can come from parameters to the annotated function. Our three parameters — `office`, `gridX`, and `gridY` — are themselves annotated with `@Path`. `@Path` says “this function parameter can be used to help assemble the relative path”. The name given as a parameter to `@Path` can be used in the relative path, wrapped in braces. Retrofit will then replace that brace-wrapped name with the actual runtime value of the parameter when we call it. So,

`"/gridpoints/{office}/{gridX},{gridY}/forecast"` will turn into something like `"/gridpoints/OKX/32,34/forecast"`, if we call `getForecast("OKX", 32, 34)`.

The US National Weather Service Web service API divides its forecasts into gridded areas served by offices. We are supplying the ID of an office (office) plus the X/Y coordinates of a particular grid cell (gridX and gridY). We will receive a forecast for that particular cell of that specific office. There is a separate Web service URL that gives us the office and cell for a given latitude and longitude. An app that provided weather information for an arbitrary location — such as one that is retrieved from LocationManager and the GPS hardware on the device — would use that URL to get the office and grid, then use the URL associated with getForecast() to get the weather. Here, we will supply the office and cell from values hard-coded in our activity, to help simplify the example.

Retrofit has a variety of additional annotations that you can add as needed. In our case, we have a @Headers annotation to add an HTTP header to our Web service call, indicating that we want a [GeoJSON](#)-encoded response.

The Repository

We have a WeatherRepository that is responsible for working with Retrofit and obtaining our weather forecast. As with some of the previous samples, the Java and Kotlin implementations diverge a bit, as Kotlin uses coroutines, while Java returns a LiveData instead.

Java

Our Java implementation of WeatherRepository looks like this:

```
package com.commonware.jetpack.weather;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;
import retrofit2.Retrofit;
import retrofit2.converter.moshi.MoshiConverterFactory;

class WeatherRepository {
    private static volatile WeatherRepository INSTANCE;
    private final NWSInterface api;

    synchronized static WeatherRepository get() {
        if (INSTANCE == null) {
            INSTANCE = new WeatherRepository();
        }

        return INSTANCE;
    }
}
```

```
private WeatherRepository() {
    Retrofit retrofit=
        new Retrofit.Builder()
            .baseUrl("https://api.weather.gov")
            .addConverterFactory(MoshiConverterFactory.create())
            .build();

    api = retrofit.create(NWSInterface.class);
}

LiveData<WeatherResult> load(String office, int gridX, int gridY) {
    final MutableLiveData<WeatherResult> result = new MutableLiveData<>();

    result.setValue(new WeatherResult(true, null, null));

    api.getForecast(office, gridX, gridY).enqueue(
        new Callback<WeatherResponse>() {
            @Override
            public void onResponse(Call<WeatherResponse> call,
                                   Response<WeatherResponse> response) {
                result.postValue(new WeatherResult(false, response.body().properties.periods, null));
            }

            @Override
            public void onFailure(Call<WeatherResponse> call, Throwable t) {
                result.postValue(new WeatherResult(false, null, t));
            }
        });

    return result;
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherRepository.java](#))

In the constructor, we get an instance of our NWSInterface from Retrofit. Specifically, we:

- Create a Retrofit.Builder object
- Provide the base URL to it (https://api.weather.gov), which will combine with the paths on the individual interface functions to assemble the entire URL to use
- Teach the Builder that it can use Moshi for converting JSON into objects, via addConverterFactory(MoshiConverterFactory.create())
- build() the resulting Retrofit object
- Call create() on the Retrofit object, to cause Retrofit to give us an instance of some generated class that implements our NWSInterface

Retrofit can handle background threading for us. Our Java implementation of NWSInterface returns our WeatherResponse wrapped in a Call object. The two main methods on Call are execute() (to perform the Web service request synchronously)

and `enqueue()` (to perform the Web service request asynchronously, on a Retrofit-supplied background thread). `load()` on `WeatherRepository` uses `enqueue()`, so Retrofit will handle our threading for us. We need to then supply a `Callback` to receive either our `WeatherResponse` or a `Throwable` if there is some problem (e.g., the Web service is down for maintenance). In our case, we wrap those results in a `WeatherResult` and update a `MutableLiveData` with that result. `WeatherResult` encapsulates an `isLoading` flag, along with our `WeatherResponse` and `Throwable` from the callback:

```
package com.commonware.jetpack.weather;

import java.util.List;

public class WeatherResult {
    final boolean isLoading;
    final List<Forecast> forecasts;
    final Throwable error;

    WeatherResult(boolean isLoading, List<Forecast> forecasts, Throwable error) {
        this.isLoading = isLoading;
        this.forecasts = forecasts;
        this.error = error;
    }
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/WeatherResult.java](#))

Hence, if we call `load()` on our `WeatherRepository` singleton, we will get a `LivData` that we can observe, where it will give us our `WeatherResult` as we progress from the loading state to either the success or failure states.

Kotlin

The Kotlin code is far simpler, because Retrofit (as of 2.6.0) has built-in support for coroutines. So, our Kotlin `NWSInterface` returns a `WeatherResponse` directly (without the `Call` wrapper), and its `getForecast()` is marked with the `suspend` keyword. So, our `WeatherRepository` can just have its own `suspend` function that delegates to Retrofit and converts the `WeatherResponse` or caught exception into `WeatherResult` objects:

```
package com.commonware.jetpack.weather

import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory
```

```
object WeatherRepository {
    private val api = Retrofit.Builder()
        .baseUrl("https://api.weather.gov")
        .addConverterFactory(MoshiConverterFactory.create())
        .build()
        .create(NWSInterface::class.java)

    suspend fun load(office: String, gridX: Int, gridY: Int) = try {
        val response = api.getForecast(office, gridX, gridY)

        WeatherResult.Content(response.properties?.periods ?: listOf())
    } catch (t: Throwable) {
        WeatherResult.Error(t)
    }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherRepository.kt](#))

In our case, then, we only have Content and Error states — `load()` returns the end result of the API call, so it has no opportunity to return a Loading state:

```
package com.commonsware.jetpack.weather

sealed class WeatherResult {
    data class Content(val forecasts: List<Forecast>) : WeatherResult()
    data class Error(val throwable: Throwable) : WeatherResult()
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherResult.kt](#))

The Motor and the View States

The Forecast objects in our WeatherResult have an integer temperature value plus a temperatureUnit string (e.g., F for Fahrenheit, C for Celsius). For display purposes, it would be nice to convert those into a single string, one that we can data bind into a TextView.

So, our MainMotor implementations will have a LiveData of MainViewState objects. MainViewState, in turn, will have a List of RowState objects, where we have a single property for the visual representation of the temperature. MainMotor will get the WeatherResult from the Web service and convert it into a MainViewState object as they come in.

Java

MainViewState and RowState look a lot like WeatherResult and Forecast, just with the single temperature field:

```
package com.commonware.jetpack.weather;

import java.util.List;

class MainViewState {
    final boolean isLoading;
    final List<RowState> forecasts;
    final Throwable error;

    MainViewState(boolean isLoading, List<RowState> forecasts, Throwable error) {
        this.isLoading = isLoading;
        this.forecasts = forecasts;
        this.error = error;
    }
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/MainViewState.java](#))

```
package com.commonware.jetpack.weather;

public class RowState {
    public final String name;
    public final String temp;
    public final String icon;

    RowState(String name, String temp, String icon) {
        this.name = name;
        this.temp = temp;
        this.icon = icon;
    }
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/RowState.java](#))

As with some of the previous examples, MainMotor uses MediatorLiveData, to fold one or more load() calls into a single results field that contains our LiveData of MainViewState objects:

```
package com.commonware.jetpack.weather;

import android.app.Application;
import java.util.ArrayList;
```

ACCESSING THE INTERNET

```
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;

public class MainMotor extends AndroidViewModel {
    final private WeatherRepository repo = WeatherRepository.get();
    final MediatorLiveData<MainViewState> results = new MediatorLiveData<>();
    private LiveData<WeatherResult> lastResult;

    public MainMotor(@NonNull Application application) {
        super(application);
    }

    void load(String office, int gridX, int gridY) {
        if (lastResult != null) {
            results.removeSource(lastResult);
        }

        lastResult = repo.load(office, gridX, gridY);
        results.addSource(lastResult, weather -> {
            ArrayList<RowState> rows = new ArrayList<>();

            if (weather.forecasts != null) {
                for (Forecast forecast : weather.forecasts) {
                    String temp =
                        getApplication().getString(R.string.temp, forecast.temperature,
                            forecast.temperatureUnit);

                    rows.add(new RowState(forecast.name, temp, forecast.icon));
                }
            }

            results.postValue(
                new MainViewState(weather.isLoading, rows, weather.error));
        });
    }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainMotor.java](#))

To create the temperature string, we use a string resource that contains placeholders for the number and unit:

```
<string name="temp">%d%s</string>
```

(from [Weather/src/main/res/values/strings.xml](#))

Then, we use `getString()` on a `Context` to retrieve that string resource and fill in those placeholders with our desired values.

Kotlin

The Kotlin implementation is similar, just using the `MutableLiveData` and `viewModelScope` approach that we saw in previous examples:

```
package com.commonware.jetpack.weather
```

```
sealed class MainViewState {  
    object Loading : MainViewState()  
    data class Content(val forecasts: List<RowState>) : MainViewState()  
    data class Error(val throwable: Throwable) : MainViewState()  
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/MainViewState.kt](#))

```
package com.commonware.jetpack.weather
```

```
data class RowState(  
    val name: String,  
    val temp: String,  
    val icon: String  
)
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/RowState.kt](#))

```
package com.commonware.jetpack.weather
```

```
import android.app.Application  
import androidx.lifecycle.AndroidViewModel  
import androidx.lifecycle.LiveData  
import androidx.lifecycle.MutableLiveData  
import androidx.lifecycle.viewModelScope  
import kotlinx.coroutines.Dispatchers  
import kotlinx.coroutines.launch
```

```
class MainMotor(application: Application) : AndroidViewModel(application) {  
    private val _results = MutableLiveData<MainViewState>()  
    val results: LiveData<MainViewState> = _results  
  
    fun load(office: String, gridX: Int, gridY: Int) {  
        _results.value = MainViewState.Loading  
  
        viewModelScope.launch(Dispatchers.Main) {  
            val result = WeatherRepository.load(office, gridX, gridY)  
        }  
    }  
}
```

```
_results.value = when (result) {
    is WeatherResult.Content -> {
        val rows = result.forecasts.map { forecast ->
            val temp = getApplication<Application>()
                .getString(
                    R.string.temp,
                    forecast.temperature,
                    forecast.temperatureUnit
                )

            RowState(forecast.name, temp, forecast.icon)
        }

        MainViewState.Content(rows)
    }
    is WeatherResult.Error -> MainViewState.Error(result.throwable)
}
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainMotor.kt](#))

The Image Loading

Eventually, our List of RowState objects makes it over to a WeatherAdapter. This is a ListAdapter that we use to fill in a RecyclerView that will show the list of forecasts.

We use data binding for the rows, where our row.xml layout has binding expressions to populate its widgets from a RowState:

```
<?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"
    xmlns:tools="http://schemas.android.com/tools">

    <data>

        <variable
            name="state"
            type="com.commonsware.jetpack.weather.RowState" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
```

```
        android:paddingBottom="8dp"
        android:paddingTop="8dp">

        <TextView
            android:id="@+id/name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{state.name}"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintBottom_toBottomOf="@id/icon"
            app:layout_constraintEnd_toStartOf="@id/temp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="@id/icon"
            tools:text="Tonight" />

        <ImageView
            android:id="@+id/icon"
            android:layout_width="0dp"
            android:layout_height="64dp"
            android:contentDescription="@string/icon"
            app:imageUrl="@{state.icon}"
            app:layout_constraintDimensionRatio="1:1"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/temp"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginEnd="16dp"
            android:layout_marginStart="16dp"
            android:text="@{state.temp}"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintBottom_toBottomOf="@id/icon"
            app:layout_constraintEnd_toStartOf="@+id/icon"
            app:layout_constraintTop_toTopOf="@id/icon"
            tools:text="72F" />

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [Weather/src/main/res/layout/row.xml](#))

In particular, our `ImageView` for the weather icon uses `app:imageUrl="@{state.icon}"` to pull the `icon` property out of the `RowState` and apply it to the `ImageView`. `ImageView` does not have an `app:imageUrl` attribute, though — we are using a `BindingAdapter` that in turn uses `Glide` to load the image:

ACCESSING THE INTERNET

```
package com.commonware.jetpack.weather;

import android.widget.ImageView;
import com.bumptech.glide.Glide;
import com.bumptech.glide.load.model.GlideUrl;
import java.util.HashMap;
import java.util.Map;
import androidx.databinding.BindingAdapter;

public class BindingAdapters {
    private static final Map<String, String> HEADERS = new HashMap<>();

    static {
        HEADERS.put("User-Agent", "me");
    }

    @BindingAdapter("imageUrl")
    public static void loadImage(ImageView view, String url) {
        if (url != null) {
            Glide.with(view.getContext())
                .load(new GlideUrl(url, () -> HEADERS))
                .into(view);
        }
    }
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/BindingAdapters.java](#))

```
package com.commonware.jetpack.weather

import android.widget.ImageView
import androidx.databinding.BindingAdapter
import com.bumptech.glide.Glide
import com.bumptech.glide.load.model.GlideUrl

private val HEADERS = mapOf("User-Agent" to "me")

@BindingAdapter("imageUrl")
fun ImageView.loadImage(url: String?) {
    url?.let {
        Glide.with(context)
            .load(GlideUrl(it) { HEADERS })
            .into(this)
    }
}
```

(from [Weather/src/main/java/com/commonware/jetpack/weather/BindingAdapters.kt](#))

For simple cases like this one, Glide has a really simple API:

- Call `Glide.with()` to get a `RequestManager` object, passing in a `Context`
- Call `load()` on the `RequestManager` to ask it to load a URL — this returns a `RequestBuilder`
- Call `into()` on the `RequestBuilder` to tell it to show the resulting image in the supplied `ImageView`

And that's it. Glide will handle doing the network I/O and populating the `ImageView` with the resulting image, using a background thread for the I/O work. It also handles recycling — if we call `into()` with an `ImageView` that already has an outstanding request, Glide will cancel the old request for us automatically.

Ordinarily, you can call `load()` with just the URL. Unfortunately, the US National Weather Service does not like the default Glide user agent header, so we have to call another version of `load()` that takes a `GlideUrl` parameter. A `GlideUrl` wraps up the details of an HTTPS request, including a custom set of headers to blend into the request.

The Results

Our `MainActivity` starts all of this off by calling `load()` on the `MainMotor`:

```
motor.load("OKX", 32, 34)
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainActivity.kt](#))

Here, `OKX`, `32`, and `34` were determined by manually invoking another Web service URL, providing the latitude and longitude for One World Trade Center in lower Manhattan.

`MainActivity` observes the `LiveData` from the `motor`, and for successful Web service calls forwards the `List` of `RowState` objects to the `WeatherAdapter` that it created and attached to a `RecyclerView`.

If you run the app, you should see an upcoming forecast for New York City:

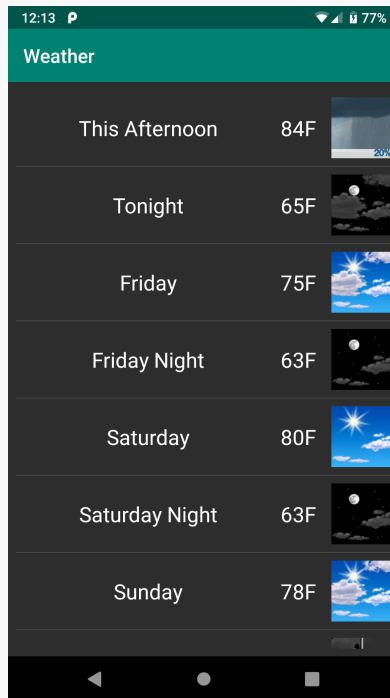


Figure 224: Weather Sample App, As Initially Launched

Here, we see that there will be a 20% chance of rain today, then clear skies for the next few days, with no signs of kaiju.

(then again, kaiju can be sneaky...)

Inverting Your Dependencies

Android app development winds up using a fair number of singletons, such as the repositories that we have used in several preceding chapters. So long as those singletons do not result in an unexpected memory leak, and so long as they are part of some organized architecture, singletons can be fine.

The problems kick in when you need different “singletons” in different situations. For example, you might need a singleton that responds in certain ways for testing, compared to the “real” singleton that you need for your production app use.

So, even though singletons are largely unavoidable in modern Android app development, how we declare and get those singletons can vary. So far, our repositories have been static fields in Java or object classes in Kotlin. Those are simple but inflexible. Dependency inversion allows us to still use singletons, yet be able to replace them as needed, such as for testing.

The Problem: Test Control

Let’s revisit the diceware samples from earlier in the book. In both `DiceLight` and `Diceware`, we have a `PassphraseRepository`. In Kotlin, it is declared as an object; in Java, it is declared as a class with a static field holding a singleton instance. And, both work, using a `SecureRandom` as the source of random numbers for use in generating the passphrase.

But now, suppose we want to test that code.

Random numbers are messy when it comes to tests. Truly random numbers are not repeatable, and so that makes our tests difficult to write. The way you test data backed by random numbers is to “seed” the random number generator, so that while

it generates random numbers, those random numbers are consistent across uses of the seed. Two random number generators started with the same seed will generate the same random numbers. Now, we have repeatable data, suitable for testing... but we do *not* want to use a fixed seed for the actual production use of the app. In the case of `SecureRandom`, it will seed itself from “sources of entropy” by default, thereby providing more truly random numbers.

So, for testing, we want a manually-seeded `SecureRandom`. For production, we want an automatically-seeded `SecureRandom`. Yet our singleton repository has just the one `SecureRandom`, and we do not have a great way in the `PassphraseRepository` to know whether we are running as part of a test or not.

We could say that the `PassphraseRepository` takes the `SecureRandom` as a parameter. Tests can pass in the seeded `SecureRandom`; production code could pass in a regular `SecureRandom`. For this limited scenario, we could make that work. However, this becomes unwieldy if we have hundreds of things that vary based on testing and need to pass those objects through several layers of app code.

The Solution: Dependency Inversion

What we want is to be able to configure `PassphraseRepository` at runtime in two different ways:

- If we are running normally, use a regular `SecureRandom`
- If we are running in a test, use a seeded `SecureRandom`

The recommended approach for this sort of pattern is to use dependency inversion. Basically, rather than `PassphraseRepository` creating a `SecureRandom` or having one passed in on every call, `PassphraseRepository` gets *created* with the desired `SecureRandom` instance. We push dependencies (`SecureRandom`) into the things that use those dependencies (`PassphraseRepository`) as part of setting up the singletons.

Dependency inversion is usually implemented using “dependency injectors”, “service locators”, or hybrid solutions. The details of the differences between those specific approaches is beyond the scope of this book — see [this Stack Overflow discussion](#) as an example of the long debates that are had regarding them.

Dependency Inversion in Android

When it comes to Android app development, there are a few dependency inversion options that have become popular.

Java: Dagger 2

The *de facto* dependency injection standard for Java app development is [Dagger](#). It is very powerful, but that power comes with a lot of confusion and complexity.

Dagger uses annotations to indicate how the dependencies get connected. In our example, annotations would indicate that `PassphraseRepository` needs a `SecureRandom` and how that `SecureRandom` gets created, both for regular code and for tests. Dagger then generates the “glue code” needed to connect those dependencies.

Kotlin: Koin and Kodein

You can use Dagger for Kotlin code as well, though the confusion and complexity increases. For Kotlin-centric projects, there are other options that are simpler to use, though they focus more on runtime dependency inversion rather than compile-time code generation.

The two most popular options for Kotlin appear to be [Kodein’s dependency injector](#) and [Koin](#). In this chapter, we will explore the use of Koin.

Koin can be used in Java projects, but it is definitely geared around use in Kotlin. As such, we will only look at Koin in a Kotlin project. The concepts of dependency inversion still hold true for Java, but the implementation in something like Dagger would look quite a bit different.

Applying Koin

With all that in mind, let’s look at the `DiceKoin` sample module in the [Sampler](#) project, to see how Koin works and what problems it solves.

The Dependency

As with most things, Koin comes from a library. Actually, Koin is made up of several

libraries, to handle different scenarios. For example, you can use Koin for non-Android projects using Kotlin.

In our case, we are using `koin-androidx-viewmodel`:

```
implementation "io.insert-koin:koin-androidx-viewmodel:2.2.3"
```

(from [DiceKoin/build.gradle](#))

This not only pulls in Koin and Koin's support for Android, but it offers specific support for the Jetpack edition of `ViewModel`, as we will see.

PassphraseRepository

`PassphraseRepository` is now a class, not an object as it was before. And, it gets a `Context` and a `SecureRandom` in its constructor. This allows us to avoid the `Context` parameter on functions like `generate()` and the locally-initialized `SecureRandom` instance:

```
package com.commonware.jetpack.diceware

import android.content.Context
import android.net.Uri
import android.util.LruCache
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import java.io.BufferedReader
import java.io.InputStream
import java.io.InputStreamReader
import java.security.SecureRandom
import java.util.*

val ASSET_URI: Uri =
    Uri.parse("file:///android_asset/eff_short_wordlist_2_0.txt")
private const val ASSET_FILENAME = "eff_short_wordlist_2_0.txt"

class PassphraseRepository(
    private val context: Context,
    private val random: SecureRandom
) {
    private val wordsCache = LruCache<Uri, List<String>>(4)

    suspend fun generate(wordsDoc: Uri, count: Int): List<String> {
        var words: List<String>?

        synchronized(wordsCache) {
            words = wordsCache.get(wordsDoc)
        }

        return words?.let { rollDemBones(it, count) }
            ?: loadAndGenerate(wordsDoc, count)
    }
}
```

INVERTING YOUR DEPENDENCIES

```
private suspend fun loadAndGenerate(wordsDoc: Uri, count: Int): List<String> =
    withContext(Dispatchers.IO) {
        val inputStream: InputStream? = if (wordsDoc == ASSET_URI) {
            context.assets.open(ASSET_FILENAME)
        } else {
            context.contentResolver.openInputStream(wordsDoc)
        }

        inputStream?.use {
            val words = it.readLines()
                .map { line -> line.split("\t") }
                .filter { pieces -> pieces.size == 2 }
                .map { pieces -> pieces[1] }

            synchronized(wordsCache) {
                wordsCache.put(wordsDoc, words)
            }

            rollDemBones(words, count)
        } ?: throw IllegalStateException("could not open $wordsDoc")
    }

private fun rollDemBones(words: List<String>, wordCount: Int) =
    List(wordCount) {
        words[random.nextInt(words.size)]
    }

private fun InputStream.readLines(): List<String> {
    val result = ArrayList<String>()

    BufferedReader(InputStreamReader(this)).forEachLine { result.add(it); }

    return result
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](https://github.com/commonsware/jetpack-diceware/blob/master/PassphraseRepository.kt))

Otherwise, `PassphraseRepository` is the same as it was, in terms of API and functionality.

MainMotor

`MainMotor` has a couple of changes as well:

- It gets a `PassphraseRepository` via its constructor, rather than referencing the former object form of the repository
- It no longer needs to pass a `Context` in its calls to the repository, so it can be a regular `ViewModel` instead of an `AndroidViewModel`

```
package com.commonsware.jetpack.diceware

import android.net.Uri
```


INVERTING YOUR DEPENDENCIES

```
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

private const val DEFAULT_WORD_COUNT = 6

class MainMotor(private val repo: PassphraseRepository) : ViewModel() {
    private val _results = MutableLiveData<MainViewState>()
    val results: LiveData<MainViewState> = _results
    private var wordsDoc = ASSET_URI

    init {
        generatePassphrase(DEFAULT_WORD_COUNT)
    }

    fun generatePassphrase() {
        generatePassphrase(
            (results.value as? MainViewState.Content)?.wordCount ?: DEFAULT_WORD_COUNT
        )
    }

    fun generatePassphrase(wordCount: Int) {
        _results.value = MainViewState.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = try {
                val randomWords = repo.generate(wordsDoc, wordCount)

                MainViewState.Content(randomWords.joinToString(" "), wordCount)
            } catch (t: Throwable) {
                MainViewState.Error(t)
            }
        }
    }

    fun generatePassphrase(wordsDoc: Uri) {
        this.wordsDoc = wordsDoc

        generatePassphrase()
    }
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/MainMotor.kt](https://github.com/commonsware/jetpack-diceware/blob/master/MainMotor.kt))

Otherwise, it too is unchanged from before.

KoinApp

At some point, though, *something* needs to be creating an instance of `PassphraseRepository`. `MainMotor` is not doing that — it is expecting something else to create the instance and supply it to the `MainMotor` constructor. Similarly, *something* needs to be creating that `SecureRandom` instance to pass to

PassphraseRepository, whenever somebody gets around to making that repository instance.

The “something” is Koin.

Compared with the Diceware sample, DiceKoin has one new class: KoinApp:

```
package com.commonware.jetpack.diceware

import android.app.Application
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.dsl.module
import java.security.SecureRandom

private val MODULE = module {
    single { SecureRandom() }
    single { PassphraseRepository(androidContext(), get()) }
    viewModel { MainMotor(get()) }
}

class KoinApp : Application() {
    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@KoinApp)
            modules(MODULE)
        }
    }
}
```

(from [DiceKoin/src/main/java/com/commonware/jetpack/diceware/KoinApp.kt](#))

Subclass of Application

We have seen the Application object before. It is a process-wide singleton instance that we can access at various points, such as calling `getApplication()` on an Activity or `AndroidViewModel`. By default, it is an instance of `android.app.Application`.

However, you can create your own subclass of `Application` and use it instead.

The big reason to do this is to get control every time your process is forked. `Application` has an `onCreate()` function, much like how an `Activity` does. `onCreate()` on an `Activity` is called when that `Activity` is created — similarly, `onCreate()` on an `Application` is called when that `Application` is created. Since the `Application` singleton is created when your process starts, your code in `onCreate()` will get called when your process starts.

As a result, we tend to use a custom `Application` subclass for cases where we need to do some process-wide initialization. Setting up dependency inversion, whether using `Koin` or something else, is usually done in a custom `Application` subclass.

Referenced in the Manifest

To tell Android to create an instance of your `Application` subclass when your process starts, you need to add an `android:name` attribute to the `<application>` element itself in the manifest:

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

    <application
        android:name=".KoinApp"
        android:allowBackup="false"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme"
        tools:ignore="GoogleAppIndexingWarning">
        <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>
```

(from [DiceKoin/src/main/AndroidManifest.xml](#))

Previously, we had only used `android:name` for things like `<activity>`. However,

`android:name` fills the same role: identifying the Java or Kotlin class that Android should use for this component.

An `<application>` element without `android:name` defaults to using `android.app.Application`. In our case, we are telling Android to use this `KoinApp` class.

Defines a Module

Part of what `KoinApp` does is define a Koin “module”. A module states what objects Koin can make available to parts of your app.

The simplest way to declare a Koin module is via the `module()` top-level function. This takes a lambda expression, in which you have statements that set up each of the Koin-managed objects (or factories of objects, as we will see).

Two of those statements are calls to `single()`:

```
single { SecureRandom() }  
single { PassphraseRepository(androidContext(), get()) }
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt](https://github.com/icekem/DiceKoin/blob/main/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt))

`single()` in a `module()` says “hey, Koin, when something asks for an object, here is one that you can use... but only have one of it for the entire app”. `single()` takes a lambda expression that creates an instance of some object, and `single()` knows what type of object that is. When something needs an object of that type, Koin can execute that lambda expression (if needed) and hand over that singleton instance.

The first `single()` call sets up a Koin-managed singleton instance of `SecureRandom`. The second `single()` call sets up a Koin-managed singleton instance of `PassphraseRepository`.

Our lambda expression for creating the `PassphraseRepository` instance needs to provide a `Context` and a `SecureRandom` to the `PassphraseRepository` constructor. For the `Context`, we use `androidContext()`, which says, “hey, Koin, you should have a `Context` that we can use here!” — we will see where that `Context` comes from shortly. For the `SecureRandom`, we use `get()`, which says “hey, Koin, search this module for a supplier of `SecureRandom` objects, and use that here!”. In our case, we set up a `SecureRandom` singleton on the preceding line. So, Koin will take that singleton `SecureRandom` and pass it to `PassphraseRepository`, when it is time to create our singleton instance of `PassphraseRepository`.

The third statement in our module is not a `single()`, but a `viewModel()`:

```
viewModel { MainMotor(get()) }
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt](#))

Courtesy of our `koin-androidx-viewmodel`, this ties into the Jetpack `ViewModel` system. Our lambda expression needs to return a `ViewModel`, and in this case it creates an instance of `MainMotor`. `MainMotor` takes a `PassphraseRepository` instance as a constructor parameter, and our `get()` call causes Koin to retrieve the `PassphraseRepository` singleton, creating it if it does not already exist.

Note that `viewModel` is not creating a singleton instance. Rather, it will defer to the Jetpack `ViewModel` system to return the proper `ViewModel` for the activity or fragment that might request one.

Starts Koin

The `onCreate()` function in `KoinApp` chains to the superclass implementation of `onCreate()`, then calls `startKoin()`:

```
override fun onCreate() {  
    super.onCreate()  
  
    startKoin {  
        androidLogger()  
        androidContext(this@KoinApp)  
        modules(MODULE)  
    }  
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt](#))

As the name suggests, `startKoin()` starts Koin. By doing that here in `onCreate()`, Koin will be ready for use in the rest of our app. Like `module()`, `startKoin()` takes a lambda expression where we can have a series of statements to describe our Koin configuration. Here, we have three:

- `androidLogger()`, telling Koin that if it has any messages to log, use `Logcat`
- `androidContext()`, telling Koin what `Context` to return from `androidContext()` calls in our module definition — in this case, we provide the `KoinApp` itself
- `modules()`, where we can provide one or more modules that we want Koin to

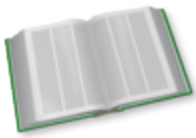
support

MainActivity

MainActivity now needs to use Koin to get its MainMotor. Previously, we used Jetpack's `viewModels()` delegate. Now, we use Koin's `viewModel()` delegate:

```
private val motor: MainMotor by viewModel()
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](https://github.com/iceknight012/dicekoin/blob/master/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt))



You can learn more about property delegates in the "Property Delegates" chapter of [Elements of Kotlin](#)!

The net effect is that the first time we try using `motor`, `viewModel()` will look in the Koin modules to see if it knows how to create an object of the desired type. In this case, we configured how to create a `MainMotor`.

For other types of objects, we would use an `inject()` property delegate instead of `viewModel()`. For example, if we wanted to directly obtain the `PassphraseRepository` in `MainActivity`, we would `inject()` it. `viewModel()`, as the name suggests, has special support for Android's `ViewModel` system.

The Dependency Inversion Flow

Here's how this works in practice, when the user taps our launcher icon.

First, `onCreate()` of `KoinApp` gets called. There, we set up our Koin module. However, none of those Koin-managed objects are actually created yet. For example, the `PassphraseRepository` is not created right away, nor is any instance of `MainMotor`.

Eventually, `MainActivity` gets instantiated. When we initialize `motor` using `viewModel()`, Koin sets up a property delegate to retrieve a `MainMotor` when needed.

We then reference `motor` in `onCreate()` of `MainActivity` as before. When we do that, Koin's property delegate tries to retrieve an existing `MainMotor` instance for this activity. That will fail, as this is the first time we are creating a `MainMotor`. So Koin executes the lambda expression that we provided to `viewModel()`, intending to

return that object as our `MainMotor` from the property delegate.

That lambda expression has a `get()` call for a parameter that needs a `PassphraseRepository`. So, Koin looks around the module and finds the `single()` for `PassphraseRepository`. Since we have not created an instance of that yet, Koin executes our lambda expression that we provided to `single()`, intending to return that object as our `PassphraseRepository` from our `get()` call.

That lambda expression in turn has a `get()` call that needs a `SecureRandom`. Once again, Koin examines the module for a match and finds the `SecureRandom` `single()`. Since we have not needed this before, Koin executes that lambda expression, saving that `SecureRandom` for future `get()` or `inject()` calls, and returns it from the `get()` call. That in turn allows the `PassphraseRepository` to be created and cached by Koin for future `get()` or `inject()` calls. And, *that* allows us to create the `MainMotor` and use it as our `ViewModel`.

If we rotate the screen and undergo a configuration change, our new `MainActivity` instance will once again use Koin to get its `MainMotor`. This time, Koin will return the previous `MainMotor` instance, courtesy of Jetpack's `ViewModel` system.

Suppose the user presses `BACK`, but then immediately re-launches our app (e.g., the `BACK` click was by accident). Now, when `MainActivity` asks Koin for a `MainMotor`, since this is a completely new activity instance, we get a completely new `MainMotor` instance. However, most likely, this is the same process as before, since the user had not been gone very long. So, we use the same `PassphraseRepository` and `SecureRandom` as before, since Koin caches them and reuses those instances, as instructed by the `single()` rule.

What This Buys Us

In the end, that's not really much additional code... but it *is* a bit more complex than what we had originally. And it may not be obvious what we gained by it, since we still wind up with all the same objects as before, just connected in a different fashion.

When it comes time to test `MainMotor` or `MainActivity`, though, that is where dependency inversion starts to become important. As noted before, we will want a controlled test instance of `SecureRandom`, one with a stable seed so we get a stable set of random numbers.

When our instrumented tests run, our `KoinApp` still gets instantiated and still

configures Koin — that does not change just because we are testing. However, after that occurs, and before we test our classes, we can *change the Koin configuration*. In particular, we can replace the self-seeding `SecureRandom` with a manually-seeded `SecureRandom`, so our tests become predictable. That code resides solely in our tests. `PassphraseRepository`, in particular, does not know about this change... because so long as it gets *some* `SecureRandom` instance, `PassphraseRepository` can do its work. So, rather than having to teach `PassphraseRepository` different rules for creating a `SecureRandom` instance (“in tests, create it this way, otherwise create it this other way”), `PassphraseRepository` *receives* a `SecureRandom`. It is our Koin configuration code — in `KoinApp` and our tests — that determine which type of `SecureRandom` we use.

We will explore this in greater detail [in an upcoming chapter](#).

Storing Data in a Room

There are three main options for storing data locally in an Android device:

- `SharedPreferences`
- SQLite databases
- Arbitrary other files or content

In this chapter, we will look at the second of those: SQLite databases. SQLite is an embedded relational database, so you can use standard SQL, while the database is a simple set of files on the device, not some server.

The Jetpack solution for working with SQLite databases is called Room. Google describes Room as providing “an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.”

In other words, Room aims to make your use of SQLite easier, through a lightweight annotation-based implementation of an [object-relational mapping \(ORM\) engine](#).

This chapter only scratches the surface of what Room has to offer, though it does include [a section on other Room features](#). Room is a fairly powerful library, and with that power comes a fair bit of complexity, once you get past the basics.

The Bookmarker sample module in the [Sampler](#) and [SamplerJ](#) projects is an app that tracks bookmarks. You will be able to share Web pages from your favorite Web browser (if it offers a “Share” option), and Bookmarker will save that URL in a Room database. It will also present the list of saved bookmarks to you, sorted by page title, and you can click on an entry in the list to view that Web page in your favorite browser.

Room Requirements

To use Room, you need two dependencies in your module's `build.gradle` file:

1. The runtime library version, using the standard `implementation` directive
2. An annotation processor, using the `annotationProcessor` directive (in Java) or the `kapt` directive (in Kotlin)

The Java edition of `Bookmarker` pulls in both of those:

```
implementation "androidx.room:room-runtime:2.3.0"
annotationProcessor "androidx.room:room-compiler:2.3.0"
```

(from [Bookmarker/build.gradle](#))

The Kotlin edition pulls in both of those, along with an additional dependency:

```
implementation "androidx.room:room-runtime:2.3.0"
implementation "androidx.room:room-ktx:2.3.0"
kapt "androidx.room:room-compiler:2.3.0"
```

(from [Bookmarker/build.gradle](#))

`androidx.room:room-runtime` is the runtime library, while `androidx.room:room-compiler` is the annotation processor. The third dependency in the Kotlin edition is `androidx.room:room-ktx`, which gives Room the ability to work with Kotlin coroutines for threading purposes.

The Kotlin edition also uses Koin for dependency inversion:

```
implementation "io.insert-koin:koin-androidx-viewmodel:2.2.3"
```

(from [Bookmarker/build.gradle](#))

Room Furnishings

Roughly speaking, your basic use of Room is divided into three sets of classes:

1. Entities, which are simple objects that model the data you are transferring into and out of the database
2. The data access object (DAO), that provides the description of the Java/Kotlin API that you want for working with certain entities
3. The database, which ties together all of the entities and DAOs for a single

SQLite database

Entities

In many ORM systems, the entity (or that system's equivalent) is a simple object that you happen to want to store in the database. It usually represents some part of your overall domain, so a payroll system might have entities representing departments, employees, and paychecks.

With Room, a better description of entities is that they are simple objects representing tables in your database, with one Java field/Kotlin property usually mapping to one column in that table.

From a coding standpoint, an entity is a class marked with the `@Entity` annotation, such as the `BookmarkEntity` class:

```
package com.commonware.jetpack.bookmarker;

import androidx.annotation.NonNull;
import androidx.room.Entity;
import androidx.room.PrimaryKey;

@Entity
public class BookmarkEntity {
    @PrimaryKey
    @NonNull
    public String pageUrl;
    public String title;
    public String iconUrl;
}
```

(from [Bookmarker/src/main/java/com/commonware/jetpack/bookmarker/BookmarkEntity.java](#))

```
package com.commonware.jetpack.bookmarker

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity
class BookmarkEntity {
    @PrimaryKey
    var pageUrl: String = ""
    var title: String? = null
    var iconUrl: String? = null
}
```

STORING DATA IN A ROOM

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkEntity.kt](#))

There is no particular superclass required for entities. The expectation is that often they will be simple objects, as we see here, where `BookmarkEntity` is a plain class with no superclass.

Each of the Kotlin properties (or Java fields) of the class will map to columns in the database. Usually, this is a 1:1 mapping (each property gets its own column), though there are ways to change that if needed. By default, the column names match the names of the properties or fields, so in this case, we have three columns:

- `pageUrl`
- `title`
- `iconUrl`

Besides the `@Entity` annotation, the only absolute requirement of an entity is that a column be designated as the primary key, usually via the `@PrimaryKey` annotation on a field or property. A primary key needs to be unique — in this case, we do not want duplicate entries for the same page URL, so we will take steps to avoid adding more than one when we add bookmarks to the database. Note that a `@PrimaryKey` needs to be non-nullable — that is a SQLite requirement that Room (and its associated Lint checks) helps to enforce.

Beyond these annotations, the rest of the code in your entity class is simply in support of the app — Room does not need anything else.

DAO

“Data access object” (DAO) is a fancy way of saying “the API into the data”. The idea is that you have a DAO that provides methods for the database operations that you need: queries, inserts, updates, deletes, whatever.

In Room, the DAO is identified by the `@Dao` annotation, applied to either an abstract class or an interface. The actual concrete implementation will be code-generated for you by the Room annotation processor.

The primary role of the `@Dao`-annotated abstract class or interface is to have one or more methods, with their own Room annotations, identifying what you want to do with the database and your entities. In the case of `Bookmarker`, we have a `BookmarkStore` interface that serves in this role.

STORING DATA IN A ROOM

```
package com.commonware.jetpack.bookmarker;

import java.util.List;
import androidx.lifecycle.LiveData;
import androidx.room.Dao;
import androidx.room.Insert;
import androidx.room.OnConflictStrategy;
import androidx.room.Query;

@Dao
public interface BookmarkStore {
    @Query("SELECT * FROM BookmarkEntity ORDER BY title")
    LiveData<List<BookmarkEntity>> all();

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void save(BookmarkEntity entity);
}
```

(from [Bookmarker/src/main/java/com/commonware/jetpack/bookmarker/BookmarkStore.java](#))

```
package com.commonware.jetpack.bookmarker

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Dao
interface BookmarkStore {
    @Query("SELECT * FROM BookmarkEntity ORDER BY title")
    fun all(): Flow<List<BookmarkEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(entity: BookmarkEntity)
}
```

(from [Bookmarker/src/main/java/com/commonware/jetpack/bookmarker/BookmarkStore.kt](#))

Most, if not all, functions on a @Dao-annotated interface will have their own Room annotations. In the case of BookmarkStore, both of the functions have a Room annotation, indicating what sort of code Room should generate for us to serve as an implementation. We do not write the SQLite access code ourselves — instead, Room handles that, while we just use the API that we declare in the @Dao.

@Query

One of our functions has a @Query annotation. Typically, these are for SQL SELECT statements, though in principle a @Query annotation can be used for any SQL. The property of the annotation contains the SQL statement to be executed. Note that this is a SQL statement and needs to use the table and column names. By default, the table name is the name of the entity class, and the column names are the names of the fields or properties, so it *looks* like you are referencing the entity itself.

Room supports a wide variety of return types for a query:

- A query can return a single entity, for cases where there should be at most one result
- A query can return a list of entities, for cases where there may be many matches
- A query can return other types than entities, for cases where your SQL does not match an entity (e.g., you are using aggregation functions like SUM)

If a query returns those sorts of types directly, then the function will be synchronous, blocking until the database I/O is completed. If, however, the query returns the type wrapped in a reactive type, then the function will perform the database I/O asynchronously — you will get the results delivered to your observer when they are ready. Also, with reactive types, if Room thinks that the data may have been altered, any active observers will get new results delivered to them automatically, without you having to call the function again.

In this case, the Java code is using LiveData as a reactive type, as that is native to the Jetpack and requires no additional libraries (e.g., RxJava). The Kotlin code is using Flow from coroutines, as that is a bit more natural in Kotlin.



You can learn more about Flow in the "Introducing Flows and Channels" chapter of [Elements of Kotlin Coroutines!](#)

@Insert, @Update, **and** @Delete

Our other function has an @Insert annotation. This performs a SQL INSERT statement, for the entity or entities provided as parameters to the function. There are also @Update and @Delete annotations, mapping to a SQL UPDATE or DELETE

statement, but BookmarkStore is not using those.

Instead, the app uses the `save()` function for both inserts and updates. This works courtesy of the `onConflict = OnConflictStrategy.REPLACE` property on the `@Insert` annotation, which says “if there already is a row in the table for this primary key, replace it with the contents of the entity”. So, when you call `save()`, either it will insert a new row or overwrite the contents of an existing row, depending on whether the `pageUrl` value on the entity is already used in the table or not.

`save()` is marked with the `suspend` keyword. The Room annotation processor will detect this and will generate a coroutine for the implementations of `save()`. Room will handle setting up the background thread for you. Without the `suspend` keyword, these functions would be synchronous, blocking until the database I/O completed.

Database

In addition to entities and DAOs, you will have at least one `@Database`-annotated abstract class, extending a `RoomDatabase` base class. This class knits together the database file, the entities, and the DAOs. In the case of Bookmarker, `BookmarkDatabase` fills this role:

```
package com.commonware.jetpack.bookmarker;

import android.content.Context;
import androidx.room.Database;
import androidx.room.Room;
import androidx.room.RoomDatabase;

@Database(entities = {BookmarkEntity.class}, version = 1)
abstract class BookmarkDatabase extends RoomDatabase {
    private static final String DB_NAME = "bookmarks.db";
    private static volatile BookmarkDatabase INSTANCE;

    synchronized static BookmarkDatabase get(Context context) {
        if (INSTANCE == null) {
            INSTANCE =
                Room.databaseBuilder(context, BookmarkDatabase.class, DB_NAME).build();
        }

        return INSTANCE;
    }
}
```


STORING DATA IN A ROOM

```
abstract BookmarkStore bookmarkStore();  
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkDatabase.java](#))

```
package com.commonsware.jetpack.bookmarker  
  
import android.content.Context  
import androidx.room.Database  
import androidx.room.Room  
import androidx.room.RoomDatabase  
  
private const val DB_NAME = "bookmarks.db"  
  
@Database(entities = [BookmarkEntity::class], version = 1)  
abstract class BookmarkDatabase : RoomDatabase() {  
  
    abstract fun bookmarkStore(): BookmarkStore  
  
    companion object {  
        @Volatile  
        private var INSTANCE: BookmarkDatabase? = null  
  
        @Synchronized  
        operator fun get(context: Context): BookmarkDatabase {  
            if (INSTANCE == null) {  
                INSTANCE =  
                    Room.databaseBuilder(context, BookmarkDatabase::class.java, DB_NAME)  
                        .build()  
            }  
  
            return INSTANCE!!  
        }  
    }  
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkDatabase.kt](#))

There are two mandatory properties on a @Database annotation:

- entities, providing a list of all of the entity classes whose tables should go into this database
- version, providing a version number for the database schema (increment this number when you ship a newer app with a newer set of entities)

Just as the @Database annotation lists the entities, each associated @Dao-annotated

class also needs to be tied into the `RoomDatabase` subclass. Specifically, you need an abstract function that returns an instance of the `@Dao`-annotated type. The name of the function can be whatever you want, as Room only cares about the return type. In a typical app, every entity and every DAO is handled by a single `RoomDatabase`, but you can have more than one if needed (e.g., one from a library and one for your own app's entities).

To retrieve an instance of the generated subclass of `BookmarkDatabase`, we need to call a function on the Room class:

- `databaseBuilder()` returns a `RoomDatabase.Builder` that will create a database in the file specified by the `filename` parameter (`DB_NAME` in the sample)
- `inMemoryDatabaseBuilder()` returns a `RoomDatabase.Builder` that will create an in-memory database, useful for your test code

`inMemoryDatabaseBuilder()` is great for testing, as it is fast and disposable. In our case, we are using `databaseBuilder()`, and using its result to set up a singleton instance of `BookmarkDatabase`.

Tying It All Together

Given all of this, your code “simply” needs to:

- Obtain an instance of your `RoomDatabase` subclass
- Call the function(s) on it to obtain your DAO objects (e.g., `bookmarkStore()`)
- Call the functions(s) on the DAO to perform database operations, including observing any `LiveData` results or calling suspend functions inside a suitable coroutine scope

We have a `BookmarkRepository` which does all of that, and a bit more, as we will explore shortly.

Other Fun Stuff in the App

Of course, this app has more to it than just a Room database, because that would not be sufficient to meet our needs. Plus, it would be really boring.

The Activity and ACTION_SEND

If you launch the app from the launcher icon (or your IDE), you will see a list of bookmarks. Initially, that list will be empty, and there is no obvious way to add bookmarks to it.

However, if you open up a Web browser on the device, browse to a page, and choose the browser's “Share” option (e.g., in an overflow menu), Bookmarker should show up as an option. If you choose it, our activity will pop up, and you should see your bookmark:

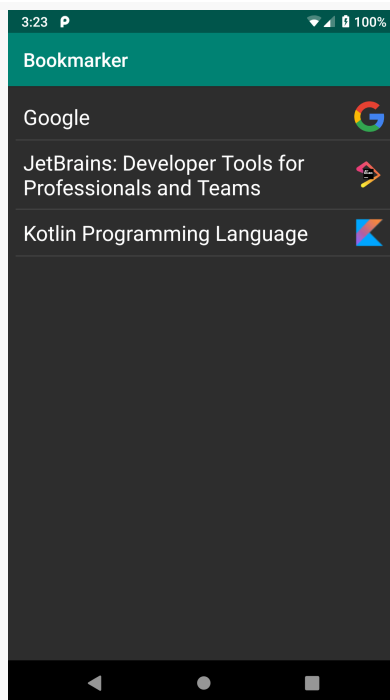


Figure 225: Bookmarker, Showing a Few Bookmarks

Our activity, in the manifest, has *two* `<intent-filter>` elements. One is the standard one to get the launcher icon. The other is for a different Intent action: `ACTION_SEND`.

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

STORING DATA IN A ROOM

```
<action android:name="android.intent.action.SEND" />
<category android:name="android.intent.category.DEFAULT" />
<data android:mimeType="text/plain" />
</intent-filter>
</activity>
```

(from [Bookmarker/src/main/AndroidManifest.xml](#))

Here we are saying that we want our activity to be started either for the standard launcher icon Intent (ACTION_MAIN and CATEGORY_LAUNCHER), but also for an Intent that has:

- ACTION_SEND as the action
- CATEGORY_DEFAULT as the category
- text/plain for the MIME type

Browsers that implement a “share” option often use that particular Intent structure, and from there, we can get the URL of the active Web page.

In our MainActivity, as part of onCreate() processing, we call a saveBookmark() method:

```
saveBookmark(getIntent());
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.java](#))

```
saveBookmark(intent)
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.kt](#))

getIntent() is how we get the Intent that was used to create this activity instance — we pass that along to saveBookmark() for processing.

saveBookmark() will try to get the URL, and if it finds one, it will pass it along to save() on our MainMotor:

```
private void saveBookmark(Intent intent) {
    if (Intent.ACTION_SEND.equals(intent.getAction())) {
        String pageUrl = getIntent().getStringExtra(Intent.EXTRA_STREAM);

        if (pageUrl == null) {
            pageUrl = getIntent().getStringExtra(Intent.EXTRA_TEXT);
        }

        if (pageUrl != null &&
```

STORING DATA IN A ROOM

```
Uri.parse(pageUrl).getScheme().startsWith("http")) {
    motor.save(pageUrl);
}
else {
    Toast.makeText(this, R.string.msg_invalid_url,
        Toast.LENGTH_LONG).show();
    finish();
}
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.java](#))

```
private fun saveBookmark(intent: Intent) {
    if (Intent.ACTION_SEND == intent.action) {
        val pageUrl = getIntent().getStringExtra(Intent.EXTRA_STREAM)
            ?: getIntent().getStringExtra(Intent.EXTRA_TEXT)

        if (pageUrl != null && Uri.parse(pageUrl).scheme!!.startsWith("http")) {
            motor.save(pageUrl)
        } else {
            Toast.makeText(this, R.string.msg_invalid_url, Toast.LENGTH_LONG).show()
            finish()
        }
    }
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.kt](#))

First, we check the action of the Intent and see if it is ACTION_SEND. If it is, we look in two “extras” of the Intent to try to find the URL:

- First, we check EXTRA_STREAM, which if it exists is always supposed to be a string representation of a Uri
- If we did not find one there, we then check EXTRA_TEXT, which could be any sort of text

In principle, for a text/plain ACTION_SEND request, we should get one of those. We then see if it looks plausible as a URL (starts with http) — if it is, we forward it to the MainMotor via save(), which in turn forwards it to a BookmarkRepository and its save() function.

The Repository

We seem to have a slight gap in our data, though. What we get from ACTION_SEND is a URL. What we have in our BookmarkEntity is a pageUrl, but also a title and an iconUrl. We need to derive values for title and iconUrl given the page URL.

Making Some (J)Soup

The way a Web browser gets the title and icon for a Web page is by parsing the HTML and looking for things like a <title> element in the <head> element. We will need to do the same thing. Fortunately, JSoup can help.

[JSoup](#) is an HTML parser for Java (and, by extension, for Kotlin/JVM). It deals with a lot of the quirks with HTML and gives us a way to navigate the document contents to find things of interest.

This app pulls in JSoup as a dependency:

```
implementation 'org.jsoup:jsoup:1.13.1'
```

(from [Bookmarker/build.gradle](#))

We then use JSoup to attempt to read in this Web page and find our title and icon. In the case of the Java implementation of BookmarkRepository, that is part of a SaveLiveData that does the work on a background thread supplied by an Executor:

```
private static class SaveLiveData extends LiveData<BookmarkResult> {
    private final String pageUrl;
    private final Executor executor;
    private final BookmarkStore store;

    SaveLiveData(String pageUrl, Executor executor, BookmarkStore store) {
        this.pageUrl = pageUrl;
        this.executor = executor;
        this.store = store;
    }

    @Override
    protected void onActive() {
        super.onActive();

        executor.execute(() -> {
            try {
                BookmarkEntity entity = new BookmarkEntity();
                Document doc = Jsoup.connect(pageUrl).get();

                entity.pageUrl = pageUrl;
                entity.title = doc.title();

                // based on https://www.mkyong.com/java/jsoup-get-favicon-from-html-page/

                String iconUrl = null;
                Element candidate = doc.head().select("link[href~=.*(ico|png)]").first();

                if (candidate == null) {
                    candidate = doc.head().select("meta[itemprop=image]").first();

                    if (candidate != null) {
```

STORING DATA IN A ROOM

```
        iconUrl = candidate.attr("content");
    }
    else {
        iconUrl = candidate.attr("href");
    }

    if (iconUrl != null) {
        URI uri = new URI(pageUrl);

        entity.iconUrl = uri.resolve(iconUrl).toString();
    }

    store.save(entity);

    postValue(new BookmarkResult(new BookmarkModel(entity), null));
}
catch (Throwable t) {
    postValue(new BookmarkResult(null, t));
}
});
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkRepository.java](#))

In the case of Kotlin, the `save()` function on `BookmarkRepository` handles it directly in a coroutine:

```
suspend fun save(pageUrl: String) =
    withContext(Dispatchers.IO) {
        val db: BookmarkDatabase = BookmarkDatabase[context]
        val entity = BookmarkEntity()
        val doc = Jsoup.connect(pageUrl).get()

        entity.pageUrl = pageUrl
        entity.title = doc.title()

        // based on https://www.mkyong.com/java/jsoup-get-favicon-from-html-page/

        val iconUrl: String? =
            doc.head().select("link[href~=.*\\.ico|png]").first()?.attr("href")
            ?: doc.head().select("meta[itemprop=image]").first().attr("content")

        if (iconUrl != null) {
            val uri = URI(pageUrl)

            entity.iconUrl = uri.resolve(iconUrl).toString()
        }

        db.bookmarkStore().save(entity)
    }
```

STORING DATA IN A ROOM

```
BookmarkModel(entity)
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkRepository.kt](#))

`JSoup.connect(pageUrl).get()` will synchronously retrieve the HTML page and parse it, throwing an exception if there is some sort of problem (e.g., cannot retrieve the page). The Document object that we get back has a `title()` function to get the page title, which is nice and easy. To get the icon URL, we have to use [some funky code, owing to multiple standards](#). In general, we use `head()` to get to the `<head>` section of the Web page, then use `select()` to use a CSS selector to find a particular element in that page, then use `attr()` to retrieve an attribute from the `first()` element matching that CSS selector. If we get a value, it could be an absolute URL (e.g., `https://somebody.com/favicon.png`) or a relative URL (e.g., `/favicon.png`). So we use `java.net.URI` to get an absolute URL given the page URL and the raw icon URL.

We then put the two URLs and the title into a `BookmarkEntity` and `save()` it using our `BookmarkStore`.

Listing the Bookmarks

`MainMotor` has a corresponding `MainViewState` that follows the loading/content/error pattern seen elsewhere in the book:

```
package com.commonsware.jetpack.bookmarker;

import java.util.List;

class MainViewState {
    final List<RowState> content;

    MainViewState(List<RowState> content) {
        this.content = content;
    }
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainViewState.java](#))

```
package com.commonsware.jetpack.bookmarker

sealed class MainViewState {
```

STORING DATA IN A ROOM

```
data class Content(val rows: List<RowState>) : MainViewState()
data class Error(val throwable: Throwable) : MainViewState()
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainViewState.kt](#))

MainMotor exposes a LiveData of MainViewState that MainActivity uses to render the RecyclerView of existing bookmarks. MainMotor uses load() on BookmarkRepository to get the bookmarks... but this returns a list of BookmarkModel objects. Our MainViewState uses a list of RowState objects to represent the data needed to render the UI for the RecyclerView rows:

```
package com.commonsware.jetpack.bookmarker;

import android.text.Spanned;
import androidx.annotation.NonNull;
import androidx.core.text.HtmlCompat;
import androidx.recyclerview.widget.DiffUtil;

public class RowState {
    public final Spanned title;
    public final String iconUrl;
    final String pageUrl;

    RowState(BookmarkModel model) {
        this.title =
            HtmlCompat.fromHtml(model.title, HtmlCompat.FROM_HTML_MODE_COMPACT);
        this.iconUrl = model.iconUrl;
        this.pageUrl = model.pageUrl;
    }

    final static DiffUtil.ItemCallback<RowState> DIFFER =
        new DiffUtil.ItemCallback<RowState>() {
            @Override
            public boolean areItemsTheSame(@NonNull RowState oldItem,
                                           @NonNull RowState newItem) {
                return oldItem == newItem;
            }

            @Override
            public boolean areContentsTheSame(@NonNull RowState oldItem,
                                           @NonNull RowState newItem) {
                return oldItem.title.toString().equals(newItem.title.toString());
            }
        };
}
```

STORING DATA IN A ROOM

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/RowState.java](#))

```
package com.commonsware.jetpack.bookmarker

import android.text.Spanned
import androidx.core.text.HtmlCompat
import androidx.recyclerview.widget.DiffUtil

class RowState(model: BookmarkModel) {
    val title: Spanned =
        HtmlCompat.fromHtml(model.title ?: "", HtmlCompat.FROM_HTML_MODE_COMPACT)
    val iconUrl = model.iconUrl
    val pageUrl = model.pageUrl

    companion object {
        val DIFFER: DiffUtil.ItemCallback<RowState> =
            object : DiffUtil.ItemCallback<RowState>() {
                override fun areItemsTheSame(
                    oldItem: RowState,
                    newItem: RowState
                ): Boolean {
                    return oldItem === newItem
                }

                override fun areContentsTheSame(
                    oldItem: RowState,
                    newItem: RowState
                ): Boolean {
                    return oldItem.title.toString() == newItem.title.toString()
                }
            }
    }
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/RowState.kt](#))

To map from `BookmarkModel` to `RowState`, in Java, `MainMotor` uses `Transformations.map()`. This takes a `LiveData` and a transformation lambda expression, one that can convert between two objects types (e.g., from a list of `BookmarkModel` to a list of `RowState`). `map()` returns another `LiveData` that emits the converted objects. In our case, we use it to map between the list of `BookmarkModel` objects and a `MainViewState` wrapping our list of `RowState` objects:

```
package com.commonsware.jetpack.bookmarker;

import android.app.Application;
import java.util.ArrayList;
import androidx.annotation.NonNull;
```

STORING DATA IN A ROOM

```
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;
import androidx.lifecycle.Transformations;

public class MainMotor extends AndroidViewModel {
    private final BookmarkRepository repo;
    private MediatorLiveData<Event<BookmarkResult>> saveEvents = new MediatorLiveData<>();
    private LiveData<Event<BookmarkResult>> lastSave;
    final LiveData<MainViewState> states;

    public MainMotor(@NonNull Application application) {
        super(application);

        repo = BookmarkRepository.get(application);
        states = Transformations.map(repo.load(),
            models -> {
                ArrayList<RowState> content = new ArrayList<>();

                for (BookmarkModel model : models) {
                    content.add(new RowState(model));
                }

                return new MainViewState(content);
            });

        LiveData<Event<BookmarkResult>> getSaveEvents() {
            return saveEvents;
        }

        void save(String pageUrl) {
            saveEvents.removeSource(lastSave);
            lastSave = Transformations.map(repo.save(pageUrl), Event::new);
            saveEvents.addSource(lastSave, event -> saveEvents.setValue(event));
        }
    }
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainMotor.java](#))

In Kotlin, BookmarkRepository is exposing a Flow rather than a LiveData. So, we can use the standard map() operator on Flow to convert our list of models into a MainViewState.Content, then use an asLiveData() extension function supplied by the Jetpack to convert that Flow into a LiveData:

```
package com.commonsware.jetpack.bookmarker

import androidx.lifecycle.*
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.launch

class MainMotor(private val repo: BookmarkRepository) : ViewModel() {
    private val _saveEvents = MutableLiveData<Event<BookmarkResult>>()
    val saveEvents: LiveData<Event<BookmarkResult>> = _saveEvents
}
```

STORING DATA IN A ROOM

```
val states: LiveData<MainViewState> = repo.load()
    .map { models -> MainViewState.Content(models.map { RowState(it) }) }
    .asLiveData()

fun save(pageUrl: String) {
    viewModelScope.launch(Dispatchers.Main) {
        _saveEvents.value = try {
            val model = repo.save(pageUrl)

            Event(BookmarkResult(model, null))
        } catch (t: Throwable) {
            Event(BookmarkResult(null, t))
        }
    }
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainMotor.kt](https://github.com/commonsware/jetpack-bookmarker/blob/master/MainMotor.kt))

What Else Does Room Offer?

Room has a wide array of other capabilities, besides the simple CRUD (create-read-update-delete) operations seen in these to-do classes, such as:

- You can add properties to the @Entity annotation to rename the table (tableName), define indexes (indices), and so on
- @Query-annotated functions that have parameters can inject those parameters into the SQL:

```
@Query("SELECT * FROM todos WHERE id = :modelId")
fun find(modelId: String): LiveData<ToDoEntity>
```

- @Insert, @Update, and @Delete functions can accept a single entity, a List of entities, or a variable argument list of entities (e.g., vararg in Kotlin)
- Using @TypeConverter and @TypeConverters annotations, you can teach Room how to save arbitrary data types into a single column, such as converting a Calendar or LocalDate into a Long or string
- Room supports @ForeignKey annotations to describe relations between two entity classes... but the entities cannot directly refer to the other entities via properties or fields. Room requires you to retrieve those related objects separately and tie them together on your own.
- When you do update your database schema by adding, changing, or removing entities, you may need to run some code to convert the older

database contents to the new schema when the user upgrades your app. Room has a Migration interface and related code to help you set up those conversions.

- In addition to LiveData and Kotlin coroutines, Room also has add-on libraries that add support for RxJava (e.g., queries returning an Observable or Single).
- By default, each SQL statement runs in its own transaction. However, through things like the @Transaction annotation, you can define your own custom transaction boundaries, where you have several SQL operations that should succeed or fail as a whole.
- Room, in conjunction with the Jetpack Paging library, supports progressive loading of large data sets, typically based on a user scrolling through a list, to minimize how much memory is needed at any one point in time.
- Room supports a pluggable database implementation. It defaults to the standard copy of SQLite that is available to any Android app, but you can replace that with some other implementation, such as [SQLCipher for Android](#) for an encrypted database.
- Room has support for SQLite’s full-text search (FTS) engine and other SQL constructs, such as views.

Note that [Elements of Android Room](#) covers Room in significantly more depth.

Examining Your Database

Your primary way of interacting with your database will be through your app. After all, that is likely to be the primary way that your *users* will interact with your database.

During development, though, it may be useful to peek at what is in the database. You have a few options for doing that.

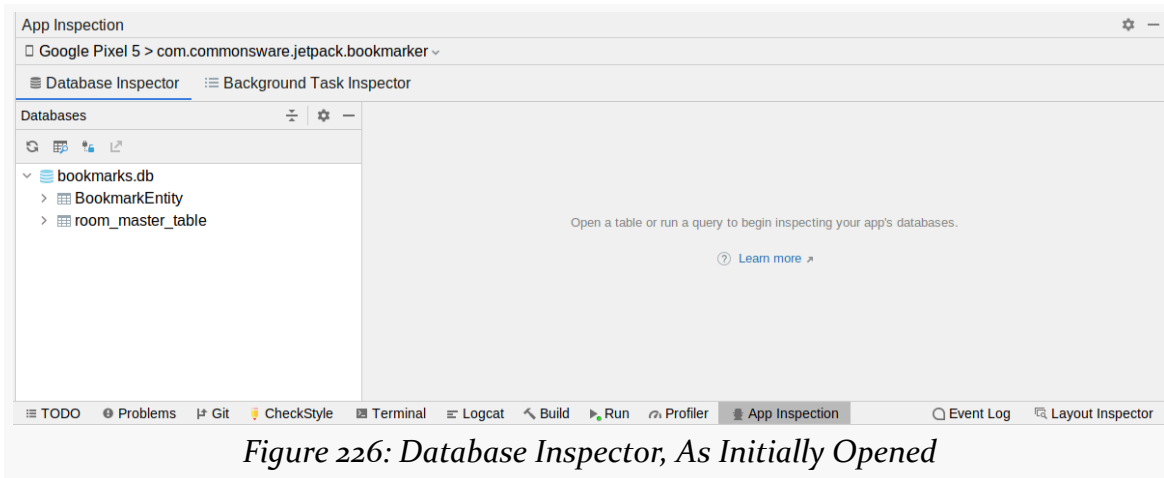
Android Studio’s Database Inspector

A leading candidate, starting with Android Studio 4.1, is Android Studio itself. There is a “Database Inspector” tool that you can use, though how you get to it varies by Android Studio version:

- For 4.1.x and 4.2.x, look for the “Database Inspector” tool, by default docked on the bottom edge
- For Arctic Fox, look for the “App Inspection” tool, by default docked on the

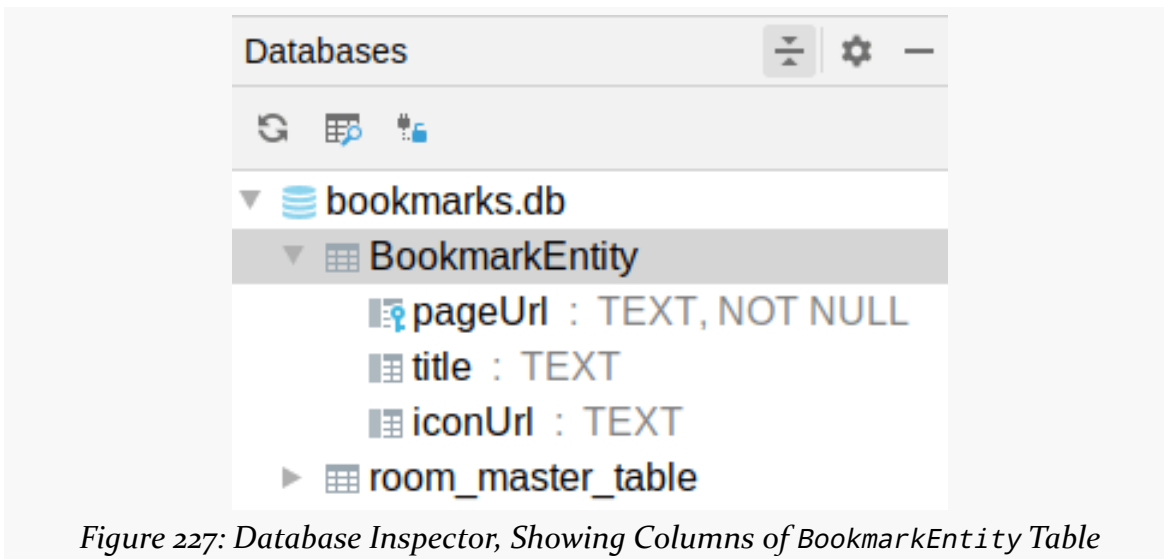
STORING DATA IN A ROOM

bottom edge — “Database Inspector” is a tab in the resulting UI:



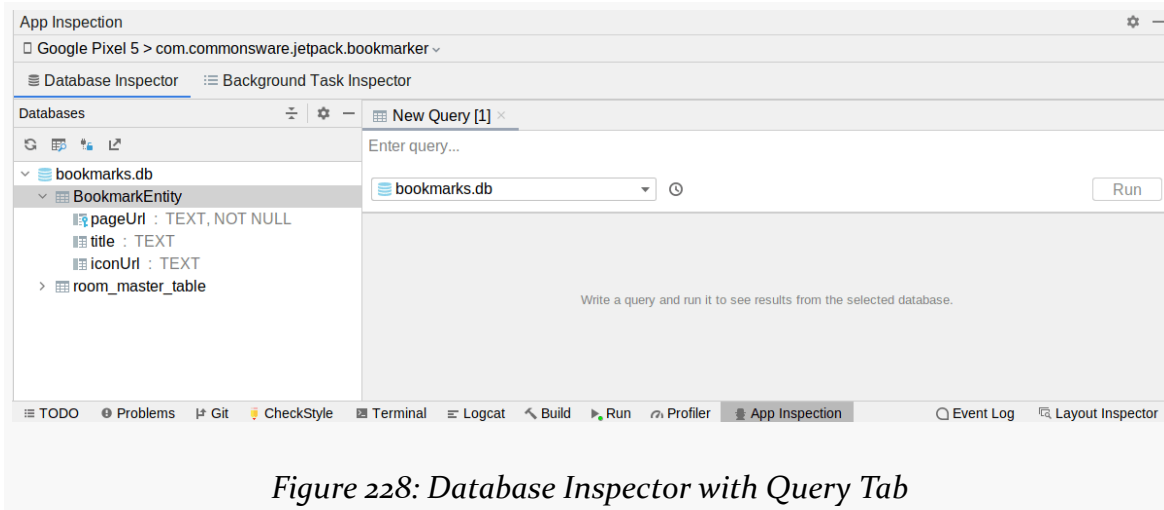
In the strip just below the title, you will see the particular app that App Inspection is offering to inspect. You can switch to something else by clicking on that entry and choosing the desired device (or emulator) and process from drop-down menus.

Database Inspector appears to look for databases in the stock location that Room and most other apps place them — in this case, it shows `bookmarks.db` in the tree on the left. Inside, it shows two tables, `BookmarkEntity` (that we defined) and `room_master_table` (that Room creates in any database that it manages). Folding open any table gives you some details of the structure:

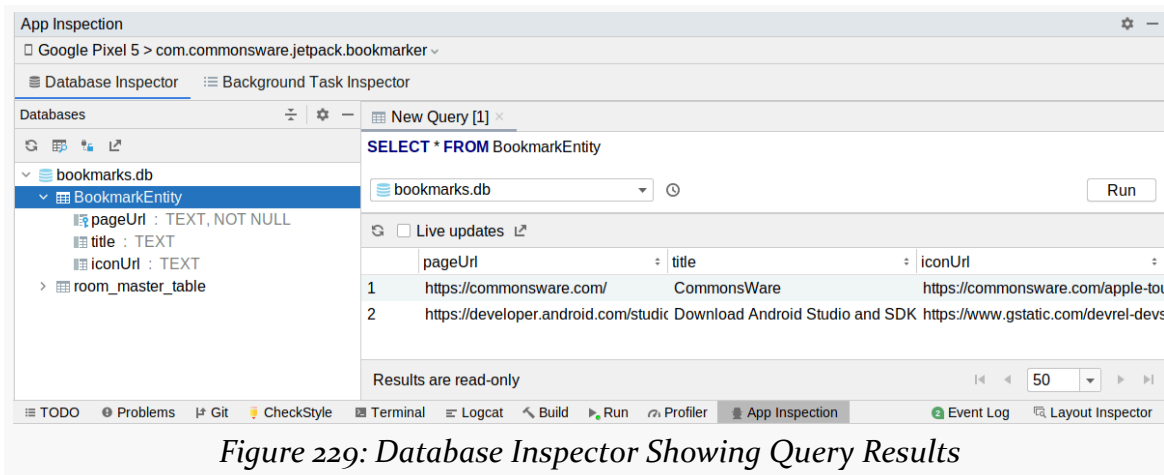


STORING DATA IN A ROOM

In the toolbar above that tree, the toolbar button that looks like a grid with a magnifying glass will open a tab for you to be able to execute queries against the selected database:



The drop-down towards the upper-left of the tab controls the database, and the field to the right is where you can enter a SQL expression. Clicking the “Run” button then executes your SQL expression, with a grid showing you the results:



If you make changes to the database via your app, you can update the Database Inspector output, either by clicking the “Refresh table” button or checking the “Live updates” checkbox. The latter will auto-refresh the query results based on database operations that your app performs.

STORING DATA IN A ROOM

Note that Database Inspector seems slow to start up and identify databases. Once those steps are completed, it runs reasonably quickly.

Other Options

Database Inspector is not your only option!

There are libraries that embed a tiny server in your debug builds, then offer some sort of client to work with that server. Facebook's [Flipper](#) is one example, built on top of Facebook's experiences with the [Stetho](#) predecessor. Stetho integrated with Chrome Dev Tools; Flipper comes with its own desktop client. With either tool, you can arrange to inspect your databases, akin to Database Inspector, without necessarily having Android Studio installed.

[Other libraries](#) are designed to expose debugging capabilities within your own app, and that may include access to your databases.

Also, SQLite is a very popular database file format, and there are many tools for working with them, such as [DB Browser for SQLite](#). You can find your database using the [Device File Explorer](#):

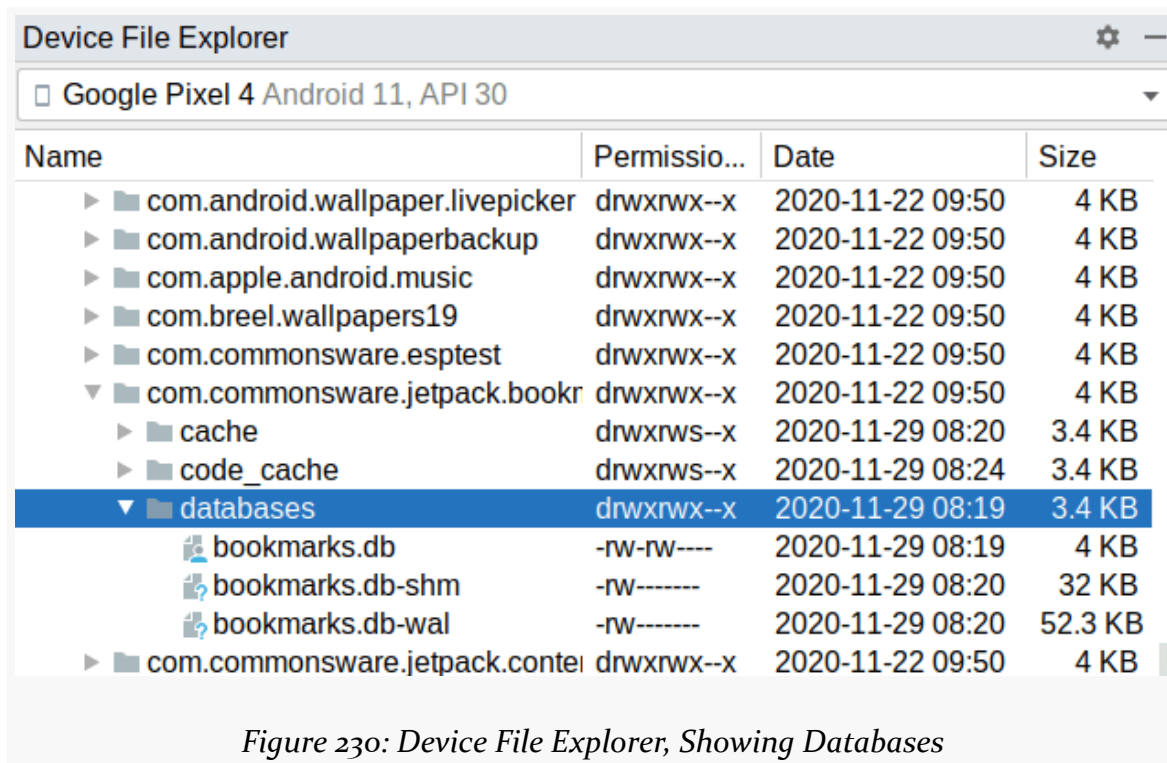


Figure 230: Device File Explorer, Showing Databases

STORING DATA IN A ROOM

The default location for Room databases is in a `databases/` directory for your app's portion of internal storage. Sometimes, you will see just the database file. Other times, such as is shown above, you will see three files:

- The database itself
- A file with the same name as the database and a `-shm` suffix
- A file with the same name as the database and a `-wal` suffix

To work with DB Browser for SQLite or similar tools, you will need to select all three of those files (if they all exist) and save them all to the same directory on your development machine.



You can learn more about examining your database using Database Inspector and other tools in the "SQLite Clients" chapter of [*Elements of Android Room*](#)!

Testing Your Changes

We saw the basics of testing in Android back in [a previous chapter](#). However, that was just with some pre-defined tests created by Android Studio's new-project wizard.

Presumably, your app will have more code that needs testing.

(if your app is purely the result of the new-project wizard templates, feel free to skip this chapter)

In this chapter, we will explore a bit more about how to write and run tests in Android, so that you can test your app code.

A Quick Recap

There are two broad categories of tests in Android: instrumented tests and unit tests.

Instrumented tests:

- Run in Android, on a device or emulator
- Have full access to the Android SDK, so you can test most of your app
- Run relatively slowly

Unit tests:

- Run outside of Android, directly on your development machine or CI server, in whatever operating system that is running (Windows, macOS, Linux)
- Have no access to the actual Android SDK, so your testing will tend to be focused on pure-Java/Kotlin code

- Run relatively quickly

Both categories of tests will look the same on the surface, in that both use JUnit4 and have similar structures (e.g., @Test-annotated methods or functions).

Which Tests Should I Write?

So, which should we use? Instrumented tests? Unit tests? Both?

If you only wanted to worry about one, choose instrumented tests. Everything can be tested using instrumented tests, while unit tests cannot readily test everything.

For larger projects — particularly those where tests will be run frequently — the speed gain from unit tests can be significant. So, a typical philosophy is:

- Test what you can with unit tests
- Test the other stuff with instrumented tests

We will return to this question [later in the chapter](#), after exploring the various testing options at our disposal.

Writing Unit Tests

First, let's look at unit tests, the ones that run on your development machine. As noted above, these reside in the test/ source set of your module.

The `ToDoTests` sample module in the [Sampler](#) contains a couple of unit tests in Kotlin. A Java project will have a similar setup, just with Java test classes.

`ToDoTests` is an expanded version of the “To Do” sample app that we explored back in the chapters on [fragments](#) and [navigation](#). It is reminiscent of the sample app being developed in [Exploring Android](#), with a variety of changes.

Configure Gradle

Gradle knows how to run unit tests “out of the box”. The only thing that you need to configure in Gradle are dependencies. Just as your app code can depend upon libraries, so too can your test code. However, instead of `implementation` (to add dependencies to your main source set), you use `testImplementation` (to add dependencies to your test source set).

TESTING YOUR CHANGES

ToDoTests has several such dependencies:

```
testImplementation "org.mockito:mockito-inline:3.12.1"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"
testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
```

(from [ToDoTests/build.gradle](#))

The two that nearly every project will use are:

- junit:junit, which contains the JUnit unit test framework
- androidx.arch.core:core-testing, which contains the main Jetpack support for unit testing

Everything else beyond that is particular to the test code that you want to write. We will see what some of these dependencies give to us later in this chapter, as we explore the tests that are already here.

Create a Test Class

For your own project, though, you will need to create your own test classes. You will not be able to download them from some book's repository.

(sorry!)

Inside the test/ source set, you can have a java/ directory containing your standard sort of Java packages and Java/Kotlin source files.

How you choose to organize your test code into packages is up to you. Note, though, that if you place test code in the same package as the code that it is testing, you can access package-private Java methods, fields, and so forth. For example, if you are testing a `foo.bar.Something` class, and your test class is `foo.bar.SomethingTest`, `SomethingTest` can access the package-private members of `Something`... even when `Something` and `SomethingTest` are in separate source sets. This gives you some amount of “white box testing”, as you can peek inside more of the objects being tested. Kotlin does not use Java's package-private system, so this distinction is less important in a pure-Kotlin project.

Your test classes are just ordinary Java/Kotlin classes. They do not have to extend

any particular base class, nor does the class necessarily have to have any annotations to teach JUnit that it is a class containing tests.

Add Test Functions

In your test class, you can add test methods or functions. These need to:

- Be annotated with `@Test`
- Take no parameters (by default)
- Be public (for Java; Kotlin functions are public by default)
- Return void (in Java) or `Unit` (the default return type in Kotlin)

In `ToDoTests`, we have a `SillyTest` test class with a pair of silly test functions:

```
package com.commonware.todo

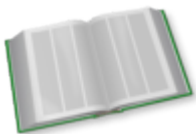
import org.junit.Assert.assertEquals
import org.junit.Test

class SillyTest {
    @Test
    fun `this is very silly`() {
        assertEquals(1, 1)
    }

    @Test
    fun thisIsEquallySillyButWithoutBackticks() {
        assertEquals(4, 2 + 2)
    }
}
```

(from [ToDoTests/src/test/java/com/commonware/todo/SillyTest.kt](#))

In Kotlin unit tests, you will sometimes find test functions that have a strange function name syntax, where the function name has spaces and/or punctuation, and the whole function name is wrapped in backticks. While this looks odd in the code, when you run the tests, since the function names become part of the test output, it allows that test output to read a bit more naturally.



You can learn more about backticks in function names in the "Escaped Method Names" chapter of [Elements of Kotlin](#)!

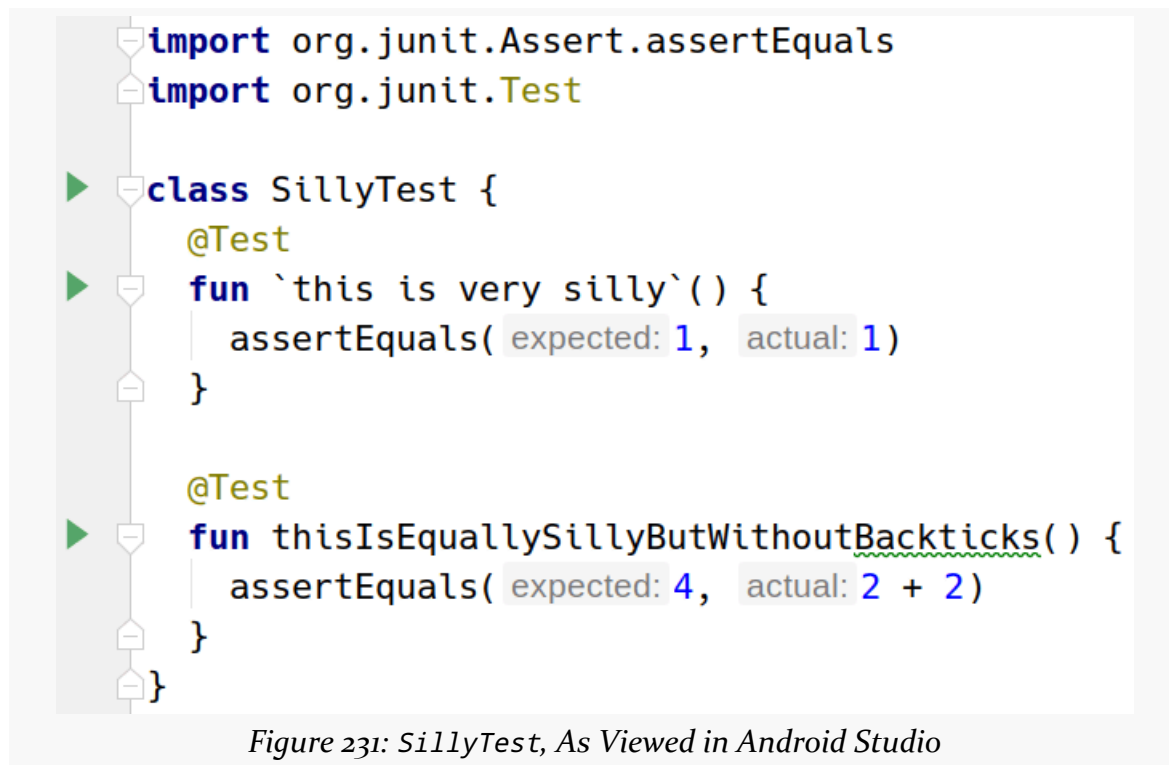
Assert Yourself

The bodies of these two test functions both call an `assertEquals()` method. This comes from JUnit, and it “does what it says on the tin”: it asserts that two values are equal. If they are, the test passes. If they are not, the test fails. If there are no assertion failures in the entire test function, the test succeeds.

There are many libraries available to add more powerful assertion rules than “does X equal Y?”. We will see one of these — Hamcrest — in the course of this chapter.

Running the Tests

The “gutter” area in the Java/Kotlin editor will show green “run” icons next to the test class and any test methods or functions:



As you might expect, clicking the “run” icon runs whatever the icon points to: an individual test method/function or the entire class.

And, as we saw [earlier in the book](#), whatever you choose to run — a single test, a single class, etc. — the results will appear in a “Run” tool, docked by default at the

TESTING YOUR CHANGES

bottom of the Android Studio window:

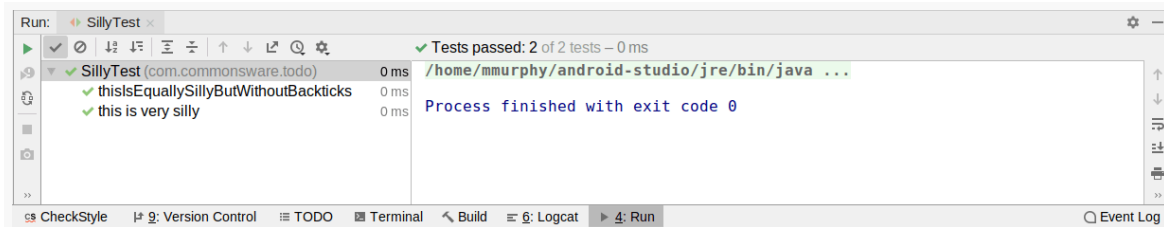


Figure 232: Android Studio Test Results

Executing Code Around the Tests

Each test method or function is executed in an independent instance of the test class. So, if we run both tests in `SillyTest`, two instances of `SillyTest` get created, one for each of the two test functions.

If you have common initialization or cleanup code that you want to execute around those test functions, you have several options:

- You can use an ordinary constructor or field/property initializers
- You can have a method or function annotated with `@Before` — this will be called before each of the test functions
- You can have a method or function annotated with `@After` — this will be called after each of the test functions
- You can have a method or function annotated with `@BeforeClass` — this will be called before any of the test functions in the test class get executed
- You can have a method or function annotated with `@AfterClass` — this will be called after any of the test functions in the test class get executed

In addition, you can have a field or property annotated with `@Rule`. A JUnit rule encapsulates “before” and “after” logic. The rule instance in the `@Rule`-annotated field or property will be able to perform some work before each test method/function and after each test method/function. This allows for easier reuse and encapsulation of whatever the before/after logic entails.

We will see examples of `@Before` and `@Rule` later in this chapter.

Employing Mocks

In unit tests, we often run into problems where we want to exercise one thing, but that thing depends upon other things that we lack:

- It depends upon the Android SDK
- It depends upon a server, and that server might not always be running at the time we would want to run the tests
- It depends upon content that does not exist in an isolated test, such as some sort of authentication token representing a logged-in user

For those things, we can use mocks.

Why Are We Being Mean?

In this case, “mock” does not mean [to ridicule something](#).

Instead, here, “mock” is used to indicate an object that pretends to be something that it is not:

- It pretends to be an Android Context
- It pretends to be your repository that ordinarily would talk to some server
- It pretends to be an authenticated user

If we set things up properly, our tests can exercise some specific functionality while using mock objects for anything that we cannot readily use in our tests. We can set up the mock objects to respond as we need them to for a given test (“stubs”). We can even verify that those mock objects were used when we expected them to be.

The Importance of Dependency Inversion

Part of the “set things up properly” is needing to ensure we have a way of getting the mocks where we need them. That is where dependency inversion comes into play.

In `ToDoTests`, we have a `ViewModel` class called `SingleModelMotor`. It works with a `ToDoRepository` to supply the UI with to-do items, plus support actions whereby users can add, edit, or delete to-do items. In the real app, `ToDoRepository` is backed by a Room-powered database. However, that does not work well in a unit test: Room expects to work with an Android Context object, and Room expects a SQLite implementation that exists on Android but not necessarily on your development

machine.

We could use a mock `ToDoRepository`... but that implies that we have a way to get the mock repository over to our `SingleModelMotor` that we are testing. If `SingleModelMotor` was referring to some singleton instance of `ToDoRepository` — such as having `ToDoRepository` be a Kotlin object — then we would be out of luck.

Instead, `ToDoTests` uses Koin for dependency inversion, as we used in [a previous chapter](#). As such, `SingleModelMotor` gets a `ToDoRepository` in its constructor, so in testing, we can just supply our mock `ToDoRepository` when we create a `SingleModelMotor` instance.

Add Mockito

The leading mock implementation in use for Android is [Mockito](#). A Java project can use Mockito directly, but a Kotlin project usually is better served by using Mockito by means of [the Mockito-Kotlin wrapper library](#).

`ToDoTests` has `testImplementation` lines in its dependencies to pull in Mockito and the Mockito-Kotlin wrapper:

```
testImplementation "org.mockito:mockito-inline:3.12.1"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"
```

(from [ToDoTests/build.gradle](#))

Define and Supply a Mock

Declaring a mock in Kotlin is very easy: call the `mock()` global function supplied by Mockito-Kotlin:

```
private val repo: ToDoRepository = mock {
    on { find(testModel.id) } doReturn flowOf(testModel)
}
```

(from [ToDoTests/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

The resulting object is not a “real” instance of `ToDoRepository`, but rather an generated object that offers the same public API.

We can then pass this mock to objects that need it, such as an instance of `SingleModelMotor`:

TESTING YOUR CHANGES

```
@Before
fun setUp() {
    underTest = SingleModelMotor(repo, testModel.id)
}
```

(from [ToDoTests/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

Here, `setUp()` will be called before each of our test functions, since it has the `@Before` annotation.

Define Stub Responses

The lambda expression in our `mock()` call defines a stub response. Our `SingleModelMotor` is going to call a `find()` function on `ToDoRepository`, and we need to indicate what our mock should return when that call is made. By default, the mock will return `null`, which is not what we want.

One Mockito-Kotlin recipe for setting up a stub uses `on()` and `doReturn()` inside of that `mock()` lambda expression. `on()` wraps the API call that we are expecting (`repo.find(testModel.id)`), where `testModel` is a test `ToDoModel` instance:

```
private val testModel = ToDoModel("this is a test")
```

(from [ToDoTests/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

The `doReturn()` function indicates the value that we want to return when that API call is made. In this case, `find()` returns a `Flow` from Kotlin coroutines, and we use the `flowOf()` top-level function to create a `Flow` wrapped around our single test model object.



You can learn more about `Flow` in the "Introducing Flows and Channels" chapter of [Elements of Kotlin Coroutines](#)!

The net result of this line is that when `SingleModelMotor` tries calling `find()` on our `ToDoRepository`, and it passes in the stated `id` value, our mock will return a `Flow` of our test model object.

Verify Calls

We can even confirm that `SingleModelMotor` actually calls `find()` as we are

TESTING YOUR CHANGES

expecting. We do that in the initial `state()` test function:

```
@Test
fun `actions pass through to repo`() {
    val replacement = testModel.copy("whatevs")

    underTest.save(replacement)
    mainDispatcherRule.dispatcher.runCurrent()

    mainDispatcherRule.dispatcher.runBlockingTest { verify(repo).save(replacement) }

    underTest.delete(replacement)
    mainDispatcherRule.dispatcher.runCurrent()

    mainDispatcherRule.dispatcher.runBlockingTest { verify(repo).delete(replacement) }
}
```

(from [ToDoTests/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

The final line of that function starts with `verify()`. This wraps a mock object with what amounts to another mock, one that is used for verification. When we call functions on the `verify()` mock, that mock confirm that the “real” mock object received a similar call during our test. If it has not, the test fails.

Using Rules

If we look at the entirety of `SimpleModelMotorTest`, we see that along with our `setUp()` function and the test function we have been examining, we have another test function... and a strange `@Rule` thing:

```
package com.commonsware.todo.ui

import com.commonsware.todo.MainDispatcherRule
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.nhaarman.mockitokotlin2.doReturn
import com.nhaarman.mockitokotlin2.mock
import com.nhaarman.mockitokotlin2.verify
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.flowOf
import kotlinx.coroutines.test.runBlockingTest
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Rule
import org.junit.Test

class SingleModelMotorTest {
    @get:Rule
    val mainDispatcherRule = MainDispatcherRule(paused = true)

    private val testModel = ToDoModel("this is a test")

    private val repo: ToDoRepository = mock {
```

TESTING YOUR CHANGES

```
on { find(testModel.id) } doReturn flowOf(testModel)
}

private lateinit var underTest: SingleModelMotor

@Before
fun setUp() {
    underTest = SingleModelMotor(repo, testModel.id)
}

@Test
fun `initial state`() {
    mainDispatcherRule.dispatcher.runCurrent()

    mainDispatcherRule.dispatcher.runBlockingTest {
        val item = underTest.states.first().item

        assertEquals(testModel, item)
    }
}

@Test
fun `actions pass through to repo`() {
    val replacement = testModel.copy("whatevs")

    underTest.save(replacement)
    mainDispatcherRule.dispatcher.runCurrent()

    mainDispatcherRule.dispatcher.runBlockingTest { verify(repo).save(replacement) }

    underTest.delete(replacement)
    mainDispatcherRule.dispatcher.runCurrent()

    mainDispatcherRule.dispatcher.runBlockingTest { verify(repo).delete(replacement) }
}
}
```

(from [ToDoTests/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

As mentioned earlier, JUnit rules represent encapsulated code that runs before and after each of our tests. In Kotlin, due to some JUnit4 quirks, we need to use `@get:Rule` syntax, saying that the `@Rule` annotation is being applied to the getter function for this property.



You can learn more about annotations on generated accessors in the "Java Interoperability" chapter of [Elements of Kotlin](#)!

Sometimes, libraries provide us with rules. For example, `InstantTaskExecutorRule` is from the Jetpack. It changes the behavior of `LiveData` such that it always uses the current thread for its work, rather than the Android main application thread. All

TESTING YOUR CHANGES

we would need to do is instantiate and annotate the `InstantTaskExecutorRule` and we get this behavior. But, this project does not use `LiveData`, so we do not need that particular rule.

Writing Rules

However, we could use a similar rule for coroutines. `SingleModelMotor` references `Dispatchers.Main`, which in Android will map to the main application thread. However, `Dispatchers.Main` has no meaning in a unit test, as we are not running on Android. We need to define what to use for `Dispatchers.Main` when running our tests, and ideally we would use something that makes the tests easier to run.



You can learn more about dispatchers in the "Choosing a Dispatcher" chapter of [Elements of Kotlin Coroutines](#)!

To that end, the `ToDoTests` project has a `MainDispatcherRule`, which we use in `SingleModelMotorTest`:

```
package com.commonware.todo

import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.test.TestCoroutineDispatcher
import kotlinx.coroutines.test.resetMain
import kotlinx.coroutines.test.setMain
import org.junit.rules.TestWatcher
import org.junit.runner.Description

// inspired by https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471

class MainDispatcherRule(paused: Boolean) : TestWatcher() {
    val dispatcher =
        TestCoroutineDispatcher().apply { if (paused) pauseDispatcher() }

    override fun starting(description: Description?) {
        super.starting(description)

        Dispatchers.setMain(dispatcher)
    }

    override fun finished(description: Description?) {
        super.finished(description)

        Dispatchers.resetMain()
        dispatcher.cleanupTestCoroutines()
    }
}
```

(from [ToDoTests/src/test/java/com/commonware/todo/MainDispatcherRule.kt](#))

TESTING YOUR CHANGES

The simplest way to write a JUnit rule is to extend `TestWatcher`, then override the `starting()` and `finishing()` functions. Those will be called before and after each test, respectively.

In the case of `MainDispatcherRule`, we:

- Create a `TestCoroutineDispatcher`
- Call `pauseDispatcher()` on the `TestCoroutineDispatcher`
- Call `Dispatchers.setMain()` before the test to have `Dispatchers.Main` point to this `TestCoroutineDispatcher`
- Call `Dispatchers.resetMain()` after the test to remove the reference to the `TestCoroutineDispatcher`
- Clean up the `TestCoroutineDispatcher` after the test

A `TestCoroutineDispatcher` is a coroutine dispatcher that operates under manual control, so our test code can indicate when it should process any queued coroutines. And, those coroutines run synchronously, on whatever thread we happen to be on. That is the behavior of `TestCoroutineDispatcher` when it is in a “paused” state — we put it in that state at the outset via the `pauseDispatcher()` call.

Then, in our `SingleModelMotorTest`, we can apply this rule using `@get:Rule` syntax. Also, we can call `runCurrent()` on the `TestCoroutineDispatcher` to indicate that it is time to run any queued coroutines.

Testing StateFlow

The `ToDoTests` project uses a mix of coroutines and `StateFlow` for its threading and reactive results. In particular, `SingleModelMotor` has a `states` property that is a `StateFlow`, emitting some viewstates that are created by means of that `find()` call on our repository.

So the overall flow of the `initial state()` test function is:

- Use `runCurrent()` to execute any coroutines that might have been invoked by creating our `SingleModelMotor` in initializers or its constructor
- Use `runBlockingTest()` to execute a coroutine that uses `first()` to get the first viewstate and confirm that it equals an expected value

Writing Instrumented Tests

Instrumented tests, to some level, work the same as unit tests:

- We use JUnit4, including annotations like `@Test`
- We can apply rules, using the same `@Rule` annotation
- We can use mocks, if desired

However, there are some differences, such as not being able to use the backtick function naming system in Kotlin. And there will be some changes to how we set up Gradle and the sorts of tests that we wind up writing.

Configure Gradle

Just as `testImplementation` statements define dependencies for unit tests (`test/`), `androidTestImplementation` statements define dependencies for instrumented tests (`androidTest/`).

Our Gradle dependencies include four such lines:

```
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
```

(from [ToDoTests/build.gradle](#))

The first one (`androidx.test:runner`) gives us our instrumentation testing core infrastructure. `androidx.test.ext:junit` and `androidx.arch.core:core-testing` provide some JUnit4 rules — notably, `androidx.arch.core:core-testing` is where utility code like that `InstantTaskExecutorRule` comes from. The final one is tied to Espresso for GUI testing, as we will explore [later in the chapter](#).

Specify the Test Runner

Earlier in the module's `build.gradle` file, we have a `testInstrumentationRunner` statement:

```
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
```

(from [ToDoTests/build.gradle](#))

This tells Android how to run our JUnit instrumented tests. JUnit uses “runner”

classes for this role, and `androidx.test.runner.AndroidJUnitRunner` is one supplied by the Jetpack that knows how to run our instrumented tests inside of an Android environment.

Identify the Test Class

With unit tests, JUnit has a default test runner that is used.

With instrumented tests, not only do we need the `testInstrumentationRunner` declaration in Gradle, but we also need to annotate our tests with a `@RunWith` annotation, further clarifying what JUnit should run and how it should run the test:

```
@RunWith(AndroidJUnit4::class)
class RosterListFragmentTest {
```

(from [ToDoTests/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

Some libraries or other frameworks may tell you to use a different `testInstrumentationRunner` or a different `@RunWith` annotation to use. For standard Jetpack tests, we use `androidx.test.runner.AndroidJUnitRunner` and `@RunWith(AndroidJUnit4::class)`, as shown here.

Access the Context

Frequently, in an instrumented test, we are using an instrumented test because something somewhere needs a `Context`.

To get a `Context` in the app being tested, your test code can use `InstrumentationRegistry.getInstrumentation().getTargetContext()` (or `InstrumentationRegistry.getInstrumentation().targetContext` in Kotlin):

```
val context = InstrumentationRegistry.getInstrumentation().targetContext
```

(from [ToDoTests/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

Note that there is also a `getContext()` method (or `context` property in Kotlin). This also returns a `Context`, but it returns one *for your `androidTest/` code*, not for the app being tested. Frequently, this is the wrong `Context` for whatever test you are trying to write.

Rewiring Koin

As noted earlier, using dependency inversion tends to make it easier for us to substitute in mocks or other replacement objects in our tests, rather than whatever code would normally be used.

In the case of the one instrumented test in `ToDoTests` — `RosterListFragmentTest` — we are using Room for a database. Room supports in-memory SQLite databases, which work just like their normal counterparts, just without the disk I/O. This executes much faster, and our data goes away once we stop referencing the database. This automatically cleans up our tests, which is very useful.

As a result, `RosterListFragmentTest` wants to replace the disk-based `ToDoDatabase` that we normally would use with an in-memory one. The `ToDoDatabase` has a `newTestInstance()` companion function that offers this:

```
package com.commonware.todo.repo

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters

private const val DB_NAME = "stuff.db"

@Database(entities = [ToDoEntity::class], version = 1)
@TypeConverters(TypeTransmogrifier::class)
abstract class ToDoDatabase : RoomDatabase() {
    abstract fun todoStore(): ToDoEntity.Store

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME).build()

        fun newTestInstance(context: Context) =
            Room.inMemoryDatabaseBuilder(context, ToDoDatabase::class.java).build()
    }
}
```

(from [ToDoTests/src/main/java/com/commonware/todo/repo/ToDoDatabase.kt](#))

However, our Koin module uses `newInstance()` and a disk-based database, not this in-memory one:

TESTING YOUR CHANGES

```
package com.commonware.todo

import android.app.Application
import com.commonware.todo.repo.ToDoDatabase
import com.commonware.todo.repo.ToDoRepository
import com.commonware.todo.ui.SingleModelMotor
import com.commonware.todo.ui.roster.RosterMotor
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.SupervisorJob
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.core.qualifier.named
import org.koin.dsl.module

class ToDoApp : Application() {
    private val koinModule = module {
        single(named("appScope")) { CoroutineScope(SupervisorJob()) }
        single { ToDoDatabase.newInstance(androidContext()) }
        single {
            ToDoRepository(
                get<ToDoDatabase>().todoStore(),
                get(named("appScope"))
            )
        }
        viewModel { RosterMotor(get()) }
        viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@ToDoApp)
            modules(koinModule)
        }
    }
}
```

(from [ToDoTests/src/main/java/com/commonware/todo/ToDoApp.kt](#))

Fortunately, Koin lets us change the module contents in our test:

```
@Before
fun setUp() {
    val context = InstrumentationRegistry.getInstrumentation().targetContext
```

TESTING YOUR CHANGES

```
val db = ToDoDatabase.newTestInstance(context)
val appScope = CoroutineScope(SupervisorJob())

repo = ToDoRepository(db.todoStore(), appScope)

loadKoinModules(module {
    single { repo }
})

runBlocking { items.forEach { repo.save(it) } }
}
```

(from [ToDoTests/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

Here, we:

- Get our Context
- Use that to get an in-memory ToDoDatabase implementation
- Wrap that in a ToDoRepository (held in a repo property)
- Call the loadKoinModules() top-level function to override our normal ToDoRepository instance with the one that we just created
- Put three test objects into that in-memory database via the repository:

```
private lateinit var repo: ToDoRepository
private val items = listOf(
    ToDoModel("this is a test"),
    ToDoModel("this is another test"),
    ToDoModel("this is... wait for it... yet another test")
)
```

(from [ToDoTests/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

Now, when our test functions execute code that obtains a ToDoRepository from Koin, they will get the in-memory repository, not the “real” one that we would normally get.

Running the Tests

The Android Studio Java/Kotlin editor will offer the same sort of “run” gutter icons as with unit tests, so you can run individual test methods or entire test classes.

TESTING YOUR CHANGES

The test results will appear in the “Run” tool in Android Studio, just like with unit tests:

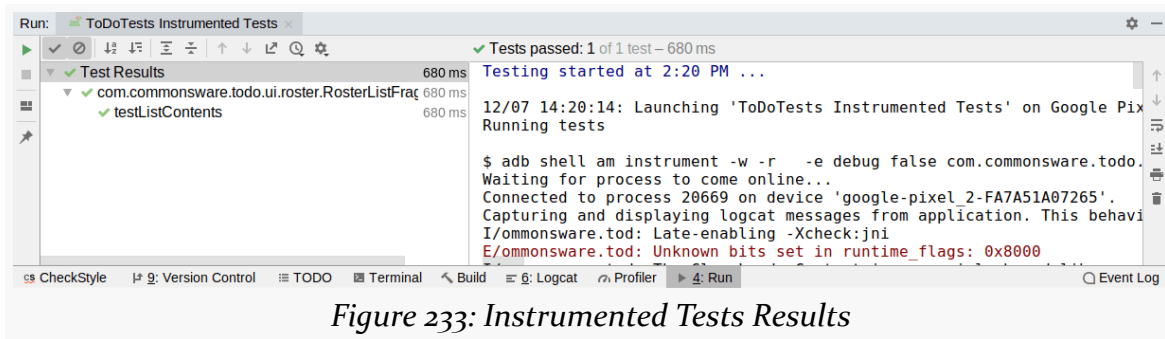


Figure 233: Instrumented Tests Results

Writing Basic Espresso Tests

Basic instrumentated 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 instrumented tests to be limiting and tedious.

Espresso is designed to simplify otherwise-complex UI testing scenarios, such as:

- Testing across screens, such as confirming that tapping a RecyclerView row in one fragment correctly launches a detail fragment 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

Espresso tests are part of your instrumented tests. Espresso simply provides another API for writing test code; those tests run alongside the rest of your instrumented tests.

However, Espresso tests are *very* fragile. They are prone to failing not due to problems in your code but due to problems when executing the tests.

Add Espresso Dependencies

One of the `androidTestImplementation` dependencies that we are pulling in is `androidx.test.espresso:espresso-core`:

TESTING YOUR CHANGES

```
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
```

(from [ToDoTests/build.gradle](#))

As you might expect, this contains the core Espresso code, and it is sufficient to write many Espresso tests. Additional libraries exist for testing things like RecyclerView.

Disable Animations

One of Espresso's limitations is that it does not like the stock animated effects that Android applies to various actions, such as launching activities. You will have better results if you disable those animations.

In the “Developer options” section of the Settings app, you will want to disable:

- Window animation scale
- Transition animation scale
- Animator duration scale

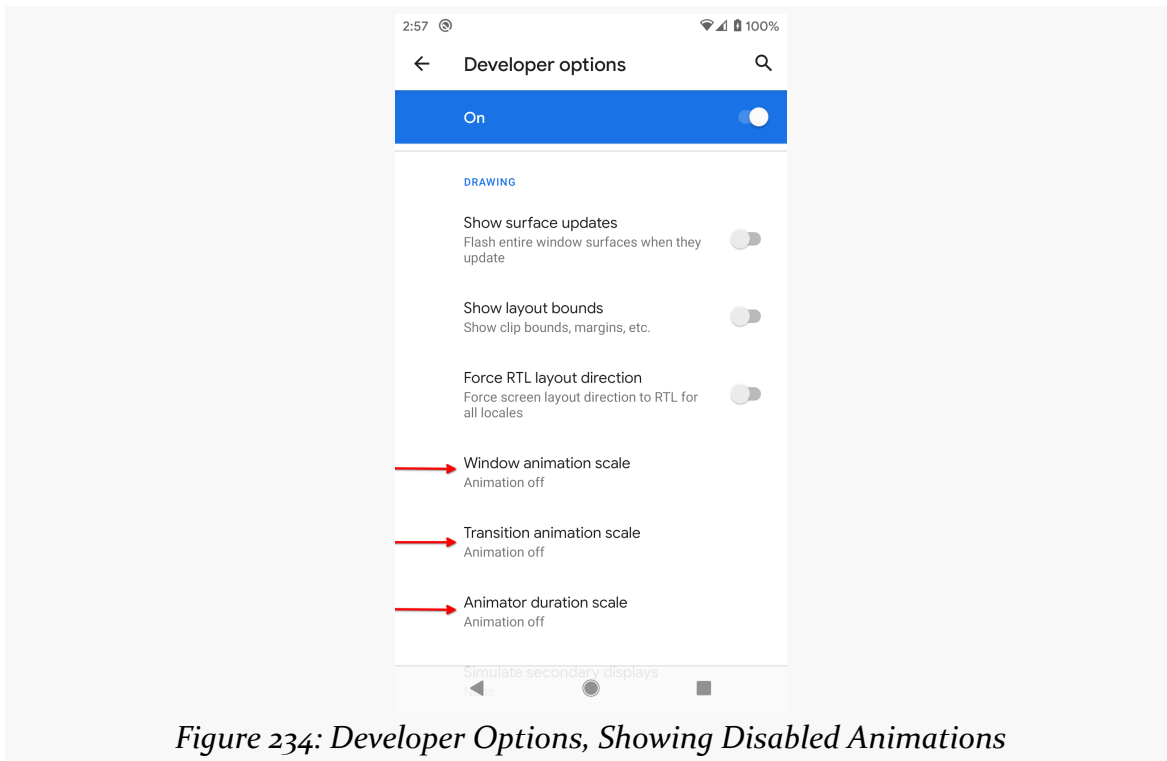


Figure 234: Developer Options, Showing Disabled Animations

Set Up the Activity or Fragment

To be able to test your UI, we need to actually have that UI appear on-screen. To that end, we can use `ActivityScenario` or `FragmentScenario`. These will launch an activity or fragment for us, allowing us to then test them afterwards.

The `testListContents()` test function in `RosterListFragmentTest` uses `ActivityScenario` to launch the `MainActivity` of this app:

```
@Test
fun testListContents() {
    ActivityScenario.launch(MainActivity::class.java)

    onView(withId(R.id.items)).check(matches(hasChildCount(3)))
}
```

(from [ToDoTests/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

`ActivityScenario.launch()` launches the activity and returns an `ActivityScenario` instance. That class implements `Closeable`, so we can use `use()` to execute our tests and close the scenario (and finish the activity) when we are done.

Find Widgets via Hamcrest Matchers

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

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.

To do that, we use an `onView()` static method supplied by Espresso. It will search the view hierarchy of the current activity for a widget.

How we identify the widget is via a “matcher”. A matcher simply is an object that can identify whether some other object matches certain criteria. In particular, Espresso uses [Hamcrest](#) matchers.

There are three main sources of matchers that you can use:

1. `ViewMatchers` 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

Here, we are using `withId()` to find some widget in `MainActivity` that has an ID of `items`.

Perform Actions

`onView()` gives us a `ViewInteraction`. 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. Actions can include things like clicking the widget (`click()`), clicking the system BACK button (`pressBack()`), swiping in a particular direction (e.g., `swipeDown()`), and so on.

In this case, we want to test whether `MainActivity` is showing our three to-do items (from the in-memory repository) in a `RecyclerView` (the `items` widget that we found). There are no actions that we need to apply here, so we skip that step.

Assert Results

Once the activity is in the desired state via any actions, we can use the `ViewInteraction` from `onView()` to validate that the widgets contain the desired content or otherwise are set up properly. 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.

The simplest `ViewAssertion` is obtained via the `matches()` static method. This takes a Hamcrest matcher and confirms that the widget matches whatever the matcher's criteria are.

TESTING YOUR CHANGES

One of the many static methods on `ViewMatchers` is `hasChildCount()`, which matches whether a `ViewGroup` (like `RecyclerView`) has a certain number of children. So, `matches(hasChildCount(3))` is a `ViewAssertion` that will succeed if the items `RecyclerView` has three children and will fail the test if not.

The net effect of the entire test is that we set up the database with three to-do items, then launch the `MainActivity` and confirm that it shows those three items:

```
package com.commonware.todo.ui.roster

import androidx.test.core.app.ActivityScenario
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.hasChildCount
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import com.commonware.todo.R
import com.commonware.todo.repo.ToDoDatabase
import com.commonware.todo.repo.ToDoModel
import com.commonware.todo.repo.ToDoRepository
import com.commonware.todo.ui.MainActivity
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.runBlocking
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.koin.core.context.loadKoinModules
import org.koin.dsl.module

@RunWith(AndroidJUnit4::class)
class RosterListFragmentTest {
    private lateinit var repo: ToDoRepository
    private val items = listOf(
        ToDoModel("this is a test"),
        ToDoModel("this is another test"),
        ToDoModel("this is... wait for it... yet another test")
    )

    @Before
    fun setUp() {
        val context = InstrumentationRegistry.getInstrumentation().targetContext
        val db = ToDoDatabase.newTestInstance(context)
        val appScope = CoroutineScope(SupervisorJob())
```



```
repo = ToDoRepository(db.todoStore(), appScope)

loadKoinModules(module {
    single { repo }
})

runBlocking { items.forEach { repo.save(it) } }
}

@Test
fun testListContents() {
    ActivityScenario.launch(MainActivity::class.java)

    onView(withId(R.id.items)).check(matches(hasChildCount(3)))
}
}
```

(from [ToDoTests/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

Another Option: UI Automator

Yet another approach for testing Android applications is UI Automator. This is designed for integration testing, seeing how your app components integrate with the rest of a device, including integrating with other applications.

UI Automator, as the name suggests, automates UIs. It simulates user input, in the form of tapping on items and the like. Tests run by UI Automator are implemented in JUnit, but 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 instrumented testing.

Wait! I Thought That Was Espresso’s Role!

Espresso also automates UI, simulating user input via JUnit tests. UI Automator tests even go inside the `androidTest/` source set and can intermingle with instrumented test code.

The difference is that UI Automator can automate the system UI (e.g., “press the HOME button”) and automate the UIs of other apps (e.g., “click the Send button in the Google Messages UI”). This is very powerful, but it also has severe limits. The actual widgets reside in another process, not your own. As a result, we have only

fairly crude forms of interacting with UIs and have even less ability to assert conditions in the other apps. Espresso gives us more control, but it can only automate our own app's UI, and even then only within a single activity.

Scenarios for UIAutomator

UI Automator is mostly used for integration testing:

- Does my activity launch properly from the launcher?
- Does my activity restart properly if I destroy it (e.g., click BACK), wind up at the home screen, then launch it again?
- My activity uses an Intent to launch an activity from a third-party app (or from the OS) — does that work correctly?

In these sorts of cases, you are not necessarily looking for a lot of details of the external activities. For example, in the last scenario, you will want to confirm that the third-party activity seems to have started and has received whatever information you are sending it (if any). But that is an activity from a third-party app, so most likely it is not your job to test all the functionality of that activity. UI Automator may suffice for your needs.

Again: What Should I Be Using?

There are a lot of testing options, including some not covered in this chapter.

(yes, this chapter could be much longer...)

There is no “right answer” for testing, so long as you test your app and feel confident that your users will not encounter bugs. That being said, a reasonably popular approach is:

- Focus most of your development efforts on testing things “behind” your UI, such as your viewmodels, repositories, data sources, and similar code. Those things frequently can be implemented as unit tests. Even if they absolutely require Android, and need to be instrumented tests, they may not need you to launch activities or fragments.
- To the greatest extent possible, refactor your application code to move as much logic out of activities and fragments as is practical. In other words, optimize your app code to allow the first bullet to cover that much more of your functionality.

TESTING YOUR CHANGES

- Resort to Espresso and UI Automator tests where needed, for logic that you simply cannot test by other means. These tend to be the most “flaky” tests. For example, your UI Automator tests might break because the third-party app you are integrating with changed, even though your app logic may still be fine.

Working with WorkManager

We have seen how to do work on the main application thread, and we have seen how to use background threads for things like disk and network I/O.

Sometimes, though, we may need to do work that happens completely independently of the UI

- The user requests to download a large file but asks for the download to be performed overnight
- We want to synchronize with a server periodically, such as every couple of hours
- The user asks us to perform some slow on-device task, such as converting a video between formats, and we want to do that work when the device is plugged in and is not being actively used by the user
- And so on

Your primary option for those scenarios is WorkManager.

The Role of WorkManager

This is not to say that WorkManager is the only solution for background work. WorkManager is designed for “deferrable” work — work that you need to have done but does not have to happen right away. This includes the possibility that the work will be done sometime after your current process has terminated, because the user left your app and a lot of time passes before it is time to do your work.

However, there are scenarios for which WorkManager is optimized. Alternative scenarios might be better handled using other techniques:

- WorkManager is designed for discrete, “transactional” tasks, not ongoing work. So, for example, WorkManager is not designed to play music continuously in the background. A “foreground service” is the solution to use for that, with background threads as needed (e.g., for disk I/O to read in the playlist details).
- WorkManager is designed for work that will happen sometime, but not at some specific time. If you need to get control at a specific time — such as to alert the user about an upcoming calendar event — use `setExactAndAllowWhileIdle()` on `AlarmManager`.
- WorkManager is designed for work that will happen eventually, but perhaps not immediately. If you have background work that has to be done in real time in response to user input (e.g., download the video that they just purchased), use a simple thread or, better yet, a reactive framework (Kotlin coroutines, RxJava, etc.). If you are concerned that the work might take too long, and your process might be terminated while that work is ongoing, use a foreground service.
- WorkManager is designed for work that might happen completely asynchronously with respect to your current process. Hence, it is not useful for cases where the work that you are doing only affects the current process, particularly its UI. So, for example, downloading avatar icons to display in your app may not make sense once the UI is gone, as you may never need those icons. For that, use a thread pool, reactive solutions (e.g., RxJava, Kotlin coroutines), or libraries that in turn use those sorts of things.

Foreground services and `AlarmManager` are powerful techniques, but ones that we will not be exploring in this particular book. [The Busy Coder's Guide to Android Development](#), while older, offers material on those subjects.

WorkManager Dependencies

The main artifact that you will use for adding WorkManager to your project is `androidx.work:work-runtime`. This contains WorkManager and its related classes.

There are a few additional artifacts that you can elect to use, of which two are general-purpose:

- `work-runtime-ktx` provides a Kotlin-specific WorkManager API
- `work-testing` is available for [testing your code that uses WorkManager](#)

Workers: They Do Work

The work that you want to have done in the background needs to be wrapped in a Worker subclass. This is an abstract class with one abstract method: `doWork()`. In that method, you put your work to be done in the background.

Note that:

- `doWork()` will keep the device awake, so you do not have to worry about the CPU powering down while your work is ongoing
- `doWork()` is called on a background thread, so you do not need to fork one yourself
- `doWork()` needs to return a `ListenableWorker.Result` object indicating if the work succeeded or failed, so `doWork()` should not be starting other background threads (directly or through libraries)
- `doWork()` cannot run forever — at best, it might run for 10 minutes before it is terminated, and it is possible that it will have less time than that

As noted above, `doWork()` returns a `ListenableWorker.Result` object. There are static factory methods on `ListenableWorker.Result` that you use to create instances. Those factory methods represent three main result scenarios:

- `success()`, which is what you are hoping for
- `retry()`, which indicates that for one reason or another you could not do the work but would like `WorkManager` to retry the work in a little while
- `failure()`, which indicates that the work could not be done and a later retry is likely to fail as well, so you are giving up

Beyond the return value and the aforementioned limitations on what you can do in `doWork()`, the actual business logic is up to you.

The `DownloadWork` sample module in the [Sampler](#) and [SamplerJ](#) projects contains a `DownloadWorker` class that downloads a file using `OkHttp` and a companion library named [Okio](#):

```
package com.commonware.jetpack.work.download;

import android.content.Context;
import android.util.Log;
import java.io.File;
import java.io.IOException;
```

WORKING WITH WORKMANAGER

```
import androidx.annotation.NonNull;
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okio.BufferedSink;
import okio.Okio;

public class DownloadWorker extends Worker {
    public static final String KEY_URL="url";
    public static final String KEY_FILENAME="filename";

    public DownloadWorker(@NonNull Context context,
                          @NonNull WorkerParameters workerParams) {
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        OkHttpClient client=new OkHttpClient();
        Request request=new Request.Builder()
            .url(getInputData().getString(KEY_URL))
            .build();

        try (Response response=client.newCall(request).execute()) {
            File dir=getApplicationContext().getCacheDir();
            File downloadedFile=
                new File(dir, getInputData().getString(KEY_FILENAME));
            BufferedSink sink=Okio.buffer(Okio.sink(downloadedFile));

            sink.writeAll(response.body().source());
            sink.close();
        }
        catch (IOException e) {
            Log.e(getClass().getSimpleName(), "Exception downloading file", e);

            return ListenableWorker.Result.failure();
        }

        return ListenableWorker.Result.success();
    }
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadWorker.java](#))

```
package com.commonware.jetpack.work.download

import android.content.Context
import android.util.Log
import androidx.work.Worker
import androidx.work.WorkerParameters
import okhttp3.OkHttpClient
import okhttp3.Request
import okio.buffer
import okio.sink
import java.io.File
import java.io.IOException

class DownloadWorker(context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams) {

    override fun doWork(): Result {
        val client = OkHttpClient()
        val request = Request.Builder()
            .url(inputData.getString(KEY_URL)!!)
            .build()

        try {
            client.newCall(request).execute().use { response ->
                val dir = applicationContext.cacheDir
                val downloadedFile = File(dir, inputData.getString(KEY_FILENAME)!!)
                val sink = downloadedFile.sink().buffer()

                response.body?.let { sink.writeAll(it.source()) }
                sink.close()
            }
        } catch (e: IOException) {
            Log.e(javaClass.simpleName, "Exception downloading file", e)

            return Result.failure()
        }

        return Result.success()
    }

    companion object {
        const val KEY_URL = "url"
        const val KEY_FILENAME = "filename"
    }
}
```

(from [DownloadWork/src/main/java/com/commonware/jetpack/work/download/DownloadWorker.kt](#))

Worker — from which DownloadWorker inherits — has a two-parameter constructor, taking a Context and a WorkerParameters object. In many cases, you can just chain to the superclass constructor, as DownloadWorker does.

Pretty much everything inside of doWork() is just application code that does the download and returns success() if the download succeeded or failure() if there was some exception during the download.

This doWork() method is using two methods that we get from Worker:

- getApplicationContext(), which works like the similarly-named method on Context, returning you the Application singleton, in case you need a Context
- getInputData(), which we will examine more closely [later in the chapter](#)

Performing Simple Work

Having a Worker is part of the puzzle. We still need to tell WorkManager to actually use that class to do our work.

If we want to perform the work once — perhaps in response to user input — we can create a OneTimeWorkRequest object to describe that work, then enqueue() it with WorkManager, as in this Java snippet:

```
OneTimeWorkRequest downloadWork=  
    new OneTimeWorkRequest.Builder(DownloadWorker.class)  
        .build();  
WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

We create a OneTimeWorkRequest via its associated Builder, which takes the Java Class object for our Worker subclass its constructor. We build() the Builder and pass the OneTimeWorkRequest to enqueue() on the WorkManager singleton, which we get by calling getInstance() on WorkManager.

After this code executes, at some point in time, an instance of DownloadWorker will be created and doWork() will be called. Exactly when that will be is indeterminate. In this particular case, *probably* it will be called fairly quickly, with doWork() being executed on a thread in a WorkManager-managed thread pool. However, it is entirely possible that the user leaves our app and our process is terminated before this work can begin. If so, WorkManager will arrange to have the work done later.

Work Inputs

However, `doWork()` would crash if we scheduled it this way. That comes back to those `getInputData()` calls from our `doWork()` method.

Often, our work needs data describing that work. In the case of `DownloadWorker`, we need to know:

- the HTTPS URL to download from
- the name of the file to download to (where the file will be placed in `getCacheDir()`)

`WorkManager` has a solution for this, via the `Data` class (perhaps named after [the android character in *Star Trek* properties](#)) (or perhaps not).

`Data` is a key-value store. The values are simple primitives plus arrays of simple primitives. We can package information into a `Data`, attach it to the work request, and then get that information from inside of `doWork()`.

Reading the `Data` is a matter of calling `getInputData()` inside of `doWork()`, then calling getter methods based on type (e.g., `getString()`). Those getter methods take the key under which the data is stored as a parameter. Getters for primitive types (e.g., `getInt()`, `getBoolean()`) also have a second parameter to use for the default response, if there is nothing associated with that key in the `Data`.

Putting `Data` into a request is a matter of creating a `Data` instance using a `Data.Builder`, which contains the corresponding setter methods (e.g., `putString()`), such as in this Java snippet:

```
OneTimeWorkRequest downloadWork=
    new OneTimeWorkRequest.Builder(DownloadWorker.class)
        .setInputData(new Data.Builder()
            .putString(DownloadWorker.KEY_URL,
                "https://commonsware.com/Android/Android-1_0-CC.pdf")
            .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
            .build())
        .build();

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

Here, we fill in the two values that the `DownloadWorker` is expecting, using `setInputData()` to attach the `Data` to our `OneTimeWorkRequest`.

Note that there is a 10KB limit on the size of the Data. Data is there mostly to provide identifiers, such as the URL and filename that we are using here. Use the Data for unique information, stuff that the Worker subclass cannot obtain from other sources (e.g., your Room database, your SharedPreferences).

Constrained Work

Frequently, the work that we want to do has some requirements. For example, in the case of DownloadWorker, it helps to have an Internet connection, as otherwise we may not be able to download the content.

WorkManager exposes a set of constraints. You can constrain your work based on:

- Whether the device has an Internet connection, or perhaps a particular type of Internet connection (e.g., an unmetered connection)
- Whether the device has a decent amount of battery life remaining, or perhaps is on a charger
- Whether the device has a decent amount of storage space available
- Whether the device is idle (so your work is less likely to interfere with the user)

To configure these, we:

- Create a Constraints.Builder,
- Call setter methods on that Builder to specify our constraints,
- build() the Builder, and
- Call setConstraints() on the request Builder to attach the constraints

This Java snippet illustrates what this might look like:

```
Constraints constraints=new Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .setRequiresBatteryNotLow(true)
    .build();
OneTimeWorkRequest downloadWork=
    new OneTimeWorkRequest.Builder(DownloadWorker.class)
        .setConstraints(constraints)
        .setInputData(new Data.Builder()
            .putString(DownloadWorker.KEY_URL,
                "https://commonsware.com/Android/Android-1_0-CC.pdf")
            .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
            .build())
```

```
.build();
```

```
WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

Here, we say that we need a network connection (of any type) and that the battery should not be low.

Tagged Work

We can also associate one or more tags with our work requests. We can later get information about our outstanding work based on tags, or cancel work based on tags.

Tags are meant to be used as categories, to identify similar pieces of work that we might want to operate on in unison:

- All downloads
- All work associated with some particular database table
- All work associated with some account
- And so on

To add tags, just call `addTag()` one or more times on the request Builder:

```
Constraints constraints = new Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .setRequiresBatteryNotLow(true)
    .build();
OneTimeWorkRequest downloadWork =
    new OneTimeWorkRequest.Builder(DownloadWorker.class)
        .setConstraints(constraints)
        .setInputData(new Data.Builder()
            .putString(DownloadWorker.KEY_URL,
                "https://commonsware.com/Android/Android-1_0-CC.pdf")
            .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
            .build())
        .addTag("download")
        .build();

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .setRequiresBatteryNotLow(true)
    .build()
val downloadWork = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
    .setConstraints(constraints)
    .setInputData(
        Data.Builder()
            .putString(
                DownloadWorker.KEY_URL,
                "https://commonsware.com/Android/Android-1_0-CC.pdf"
            )
            .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
            .build()
    )
    .addTag("download")
    .build()

WorkManager.getInstance(getApplication()).enqueue(downloadWork)
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

Here, we use `addTag()` to tag this work as download.

Monitoring Work

WorkManager does not provide a built-in means for you to monitor progress inside of an individual piece of work. It does, however, provide you with an API for monitoring the gross state changes of a piece of work: is it enqueued, is it running, is it completed, etc.

Getting the Status Updates

To find out about the general state changes in the life of a piece of work, you can use `getWorkInfoByIdLiveData()`, available on WorkManager. Each request has an ID, generated by the WorkManager system, which you get by calling `getId()` on the request:

```
final LiveData<WorkInfo> liveOpStatus =
    WorkManager.getInstance(getApplication()).getWorkInfoByIdLiveData(
        downloadWork.getId());
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
val liveOpStatus =  
    WorkManager.getInstance(getApplication())  
        .getWorkInfoByIdLiveData(downloadWork.id)
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

The LiveData that we get back will emit WorkInfo updates for the work identified by this ID. A WorkInfo, in turn, holds a State enum, that indicates what phase of the WorkManager process this piece of work is in:

- ENQUEUED
- BLOCKED (for use with [chained work](#))
- RUNNING
- SUCCEEDED
- FAILED
- CANCELED (for use with [canceling work](#))

You can then arrange to observe the LiveData or otherwise make use of its updates.

Consuming the Status Updates... In Code

The code shown in this chapter so far that created the OneTimeWorkRequest and enqueued the work is in a DownloadViewModel:

```
package com.commonsware.jetpack.work.download;  
  
import android.app.Application;  
import androidx.annotation.NonNull;  
import androidx.lifecycle.AndroidViewModel;  
import androidx.lifecycle.LiveData;  
import androidx.lifecycle.MediatorLiveData;  
import androidx.work.Constraints;  
import androidx.work.Data;  
import androidx.work.NetworkType;  
import androidx.work.OneTimeWorkRequest;  
import androidx.work.WorkInfo;  
import androidx.work.WorkManager;  
  
public class DownloadViewModel extends AndroidViewModel {  
    public final MediatorLiveData<WorkInfo> liveWorkStatus =  
        new MediatorLiveData<>();  
  
    public DownloadViewModel(@NonNull Application application) {  
        super(application);  
    }  
}
```

```
public void doTheDownload() {
    Constraints constraints = new Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED)
        .setRequiresBatteryNotLow(true)
        .build();
    OneTimeWorkRequest downloadWork =
        new OneTimeWorkRequest.Builder(DownloadWorker.class)
            .setConstraints(constraints)
            .setInputData(new Data.Builder()
                .putString(DownloadWorker.KEY_URL,
                    "https://commonsware.com/Android/Android-1_0-CC.pdf")
                .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
                .build())
            .addTag("download")
            .build();

    WorkManager.getInstance(getApplication()).enqueue(downloadWork);

    final LiveData<WorkInfo> liveOpStatus =
        WorkManager.getInstance(getApplication()).getWorkInfoByIdLiveData(
            downloadWork.getId());

    liveWorkStatus.addSource(liveOpStatus, workStatus -> {
        liveWorkStatus.setValue(workStatus);

        if (workStatus.getState().isFinished()) {
            liveWorkStatus.removeSource(liveOpStatus);
        }
    });
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
package com.commonsware.jetpack.work.download

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.MediatorLiveData
import androidx.work.*

class DownloadViewModel(application: Application) :
    AndroidViewModel(application) {
    val liveWorkStatus = MediatorLiveData<WorkInfo>()

    fun doTheDownload() {
        val constraints = Constraints.Builder()
```

```
.setRequiredNetworkType(NetworkType.CONNECTED)
.setRequiresBatteryNotLow(true)
.build()
val downloadWork = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
    .setConstraints(constraints)
    .setInputData(
        Data.Builder()
            .putString(
                DownloadWorker.KEY_URL,
                "https://commonsware.com/Android/Android-1_0-CC.pdf"
            )
            .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
            .build()
    )
    .addTag("download")
    .build()

WorkManager.getInstance(getApplication()).enqueue(downloadWork)

val liveOpStatus =
    WorkManager.getInstance(getApplication())
        .getWorkInfoByIdLiveData(downloadWork.id)

liveWorkStatus.addSource(liveOpStatus) { workStatus ->
    liveWorkStatus.value = workStatus

    if (workStatus.state.isFinished) {
        liveWorkStatus.removeSource(liveOpStatus)
    }
}
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

The `doTheDownload()` method will be called when the user clicks a button in the UI of `MainActivity`. That triggers our creation of the work request.

`DownloadViewModel` takes the `MediatorLiveData` approach seen elsewhere in the book. Consumers of the `DownloadViewModel`, such as our `MainActivity`, have access to a `liveWorkStatus` field that represents the outbound stream of work status updates. For each `doTheDownload()` call, we chain the `LiveData` for this individual download onto the `MediatorLiveData`, removing it as a source once the `State` reaches a terminal condition (`isFinished()`), which will be true for a `State` of `SUCCEEDED`, `FAILED`, or `CANCELED`.

WORKING WITH WORKMANAGER

The result is that our MainActivity can observe liveWorkStatus, without having to worry about individual LiveData objects from individual download requests.

MainActivity observes liveWorkStatus and uses it to display a Toast when the download is finished:

```
package com.commonware.jetpack.work.download;

import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
import com.commonware.jetpack.work.download.databinding.ActivityMainBinding;
import androidx.appcompat.app.AppCompatActivity;
import androidx.databinding.BindingAdapter;
import androidx.lifecycle.ViewModelProvider;
import androidx.work.WorkInfo;

public class MainActivity extends AppCompatActivity {
    @BindingAdapter("android:enabled")
    public static void setEnabled(View v, WorkInfo info) {
        if (info==null) {
            v.setEnabled(true);
        }
        else {
            v.setEnabled(info.getState().isFinished());
        }
    }

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final DownloadViewModel vm= new ViewModelProvider(this).get(DownloadViewModel.class);

        binding=ActivityMainBinding.inflate(getLayoutInflater());
        binding.setViewModel(vm);
        binding.setLifecycleOwner(this);

        setContentView(binding.getRoot());

        vm.liveWorkStatus.observe(this, workStatus -> {
            if (workStatus!=null && workStatus.getState().isFinished()) {
                Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show();
            }
        });
    }
}
```

(from [DownloadWork/src/main/java/com/commonware/jetpack/work/download/MainActivity.java](#))

```
package com.commonware.jetpack.work.download

import android.os.Bundle
import android.view.View
```

```
import android.widget.Toast
import androidx.activity.viewModels
import androidx.appcompat.app.AppCompatActivity
import androidx.databinding.BindingAdapter

import androidx.work.WorkInfo
import com.commonware.jetpack.work.download.databinding.ActivityMainBinding

@BindingAdapter("android:enabled")
fun View.setEnabled(info: WorkInfo?) {
    isEnabled = info?.state?.isFinished ?: true
}

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val vm: DownloadViewModel by viewModels()
        val binding = ActivityMainBinding.inflate(layoutInflater)

        binding.viewModel = vm
        binding.lifecycleOwner = this

        setContentView(binding.root)

        vm.liveWorkStatus.observe(this) { workStatus ->
            if (workStatus != null && workStatus.state.isFinished) {
                Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show()
            }
        }
    }
}
```

(from [DownloadWork/src/main/java/com/commonware/jetpack/work/download/MainActivity.kt](#))

Consuming the Status Updates... In Data Binding

MainActivity — and its activity_main layout resource — use data binding. Partially, this is to get control to DownloadViewModel when the user clicks a button. But we also want to disable the button while the download is going on, to reduce the likelihood of accidentally triggering multiple downloads.

Data binding has dedicated support for LiveData. If you have a LiveData available through a <variable>, you can reference the LiveData in a binding expression as if it were a simple variable representing the data. The data binding framework will

take care of the details of observing the LiveData and updating your UI when the data changes.

To make this work, our layout has a reference to the DownloadViewModel and has a binding expression on android:enabled that looks at the state:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>

    <data>

        <variable
            name="viewModel"
            type="com.commonware.jetpack.work.download.DownloadViewModel" />
    </data>

    <androidx.constraintlayout.widget.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=".MainActivity">

        <Button
            android:id="@+id/download"
            android:layout_width="0dp"
            android:layout_height="0dp"
            android:text="@string/btn_title"
            android:onClick="@{() -> viewModel.doTheDownload()}"
            android:enabled="@{viewModel.liveWorkStatus}"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [DownloadWork/src/main/res/layout/activity_main.xml](#))

Our MainActivity then does three things in support of all of this:

1. It has a BindingAdapter that can update the enabled status of a view given a WorkInfo, allowing us to use a binding expression on android:enabled
2. It binds the DownloadViewModel into the binding
3. It calls setLifecycleOwner() on the binding, which the data binding

framework will use for observing the LiveData:

```
package com.commonware.jetpack.work.download;

import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
import com.commonware.jetpack.work.download.databinding.ActivityMainBinding;
import androidx.appcompat.app.AppCompatActivity;
import androidx.databinding.BindingAdapter;
import androidx.lifecycle.ViewModelProvider;
import androidx.work.WorkInfo;

public class MainActivity extends AppCompatActivity {
    @BindingAdapter("android:enabled")
    public static void setEnabled(View v, WorkInfo info) {
        if (info==null) {
            v.setEnabled(true);
        }
        else {
            v.setEnabled(info.getState().isFinished());
        }
    }

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final DownloadViewModel vm= new ViewModelProvider(this).get(DownloadViewModel.class);

        binding=ActivityMainBinding.inflate(getLayoutInflater());
        binding.setViewModel(vm);
        binding.setLifecycleOwner(this);

        setContentView(binding.getRoot());

        vm.liveWorkStatus.observe(this, workStatus -> {
            if (workStatus!=null && workStatus.getState().isFinished()) {
                Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show();
            }
        });
    }
}
```

(from [DownloadWork/src/main/java/com/commonware/jetpack/work/download/MainActivity.java](#))

```
package com.commonware.jetpack.work.download

import android.os.Bundle
import android.view.View
import android.widget.Toast
import androidx.activity.viewModels
import androidx.appcompat.app.AppCompatActivity
import androidx.databinding.BindingAdapter
```

```
import androidx.work.WorkInfo
import com.commonware.jetpack.work.download.databinding.ActivityMainBinding

@BindingAdapter("android:enabled")
fun View.setEnabled(info: WorkInfo?) {
    isEnabled = info?.state?.isFinished ?: true
}

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val vm: DownloadViewModel by viewModels()
        val binding = ActivityMainBinding.inflate(layoutInflater)

        binding.viewModel = vm
        binding.lifecycleOwner = this

        setContentView(binding.root)

        vm.liveWorkStatus.observe(this) { workStatus ->
            if (workStatus != null && workStatus.state.isFinished) {
                Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show()
            }
        }
    }
}
```

(from [DownloadWork/src/main/java/com/commonware/jetpack/work/download/MainActivity.kt](#))

Here, we map the State from a WorkInfo to the boolean value to use for the android:enabled attribute. Basically, if the WorkInfo is null or is finished, the button is enabled, otherwise it is disabled. So, as the LiveData emits new WorkInfo objects, data binding takes each, calls this `setEnabled()` method, and uses that to update the enabled state of the Button.

Canceling Work

Frequently, the work that we enqueue into the WorkManager is “fire and forget”, where either the work succeeds or fails on its own. Occasionally, though, we may need to try to cancel a piece of enqueued work. For example, we might offer a cancel button in the UI to allow the user to abandon some enqueued request (e.g., do not download the thing that the user just requested to download).

For that, you can call `cancelWorkById()` or `cancelAllWorkByTag()` on `WorkManager`. The former takes the work's ID (from `getId()` on the `WorkRequest`), while the latter takes a tag. Since IDs are unique, `cancelWorkById()` will only try to cancel that one piece of work, while `cancelAllWorkByTag()` will try to cancel all enqueued work associated with that tag. So, for example, if you associate the download tag with your download work requests, `cancelAllWorkByTag("download")` will try to cancel all of those requests.

Note, though, that cancellation is “best effort”. In particular, if the work has already begun, it might not be canceled. In some cases, this is fine. In other cases, you might want to both cancel the work in `WorkManager` and take steps to ensure that any affected running work finds out about the cancellation. For example, you might have some state field somewhere that the work can monitor to see if it should continue doing whatever it is doing.

Delayed Work

Typically, we are happy to have our work begin right away, if the conditions allow it. Occasionally, we may want to intentionally delay that work for a bit.

For this, you can call `setInitialDelay()` on the `OneTimeWorkRequest.Builder` as part of configuring the work. There are two flavors of `setInitialDelay()`:

- one takes a long value and a `TimeUnit`, where the `TimeUnit` indicates what unit of measure the long is in (e.g., `TimeUnit.SECONDS`)
- one takes a `Duration`, which is part of Java 8 and only available on API Level 26 and higher

In either case, the work will be delayed by at least that amount of time. However, depending on circumstances (constraints, Doze mode, etc.), the work might happen substantially later than the delay period. Do not assume that your work will start immediately after your delay period.

Parallel Work

Suppose you have *N* pieces of work to be done in (approximate) parallel. As it turns out, `enqueue()` on `WorkManager` takes either a varargs of `WorkRequest` or a `List` of `WorkRequest` objects. If you pass in more than one `WorkRequest`, all will be enqueued, and all will run when possible, based upon their constraints and the number of threads in the `WorkManager` thread pool.

WorkManager has a default thread pool, and in many cases it will be sufficient for your needs. If you wish to control that thread pool, call the static `initialize()` on `WorkManager` *once*, such as from `onCreate()` of a custom `Application`. `initialize()` takes a `Context` and a `Configuration`. You get a `Configuration` through the builder pattern:

- Create an instance of `Configuration.Builder`
- Call `setExecutor()` on that `Builder` with an `Executor` that will serve as the thread pool for the `WorkManager`
- Call `build()` on the `Builder` to get the `Configuration`

Chained Work

Where `WorkManager` shines is in its support for chained work. Chained work is where you set up work requests that in turn depend upon other work requests. Later work requests in the chain are only performed if the previous ones succeeded. And, work requests can supply data to the next request in the chain, akin to command-line pipelines or basic workflow systems.

Why?

On the one hand, chained work may not seem necessary. In principle, what you do as a series of work requests could be done in one large work request.

The big benefit of splitting the work into separate requests comes with the application of constraints. For example, the sample app that we will examine demonstrates chained work by downloading a ZIP file, then unZIPping it. Downloading a ZIP file requires an Internet connection, but unZIPping it does not. By providing separate constraints for each work request, you can require a network connection for the download, yet not require it for the unZIP task, thereby allowing that work to proceed even if Internet connectivity is lost.

Also, smaller `Worker` classes can be made more reusable. One can imagine a library of common `Worker` classes. Rather than having to write your own `CompositeWorker` that used several `Worker` classes, you can simply set up a chain using existing APIs.

Chained work also helps to address the delivery of status updates as a larger task is being processed. Each `WorkRequest` in the chain has its own `WorkStatus` that can be monitored via `LiveData`. This way, you can at least get coarse-grained information about how the chain overall is proceeding.

How Do We Chain Work?

To enqueue a `WorkRequest`, we used `enqueue()` on the `WorkManager` instance. In truth, that is a convenience method. This:

```
WorkManager.getInstance(getApplicationContext())  
    .enqueue(request);
```

is really this:

```
WorkManager.getInstance(getApplicationContext())  
    .beginWith(request)  
    .enqueue();
```

`beginWith()` returns a `WorkContinuation`. This is an object that knows a `WorkRequest` to process and knows how to be chained.

To have a follow-on `WorkRequest` in a simple two-element chain, call `then()` on the `WorkContinuation` before the terminal `enqueue()` call:

```
WorkManager.getInstance(getApplicationContext())  
    .beginWith(request)  
    .then(otherRequest)  
    .enqueue();
```

Now, `request` will be processed, and if it succeeds, `then` (and only then) will `otherRequest` be processed.

How Do We Pass Data Along the Chain?

We provide input to a `WorkRequest` via its `Builder` and `setInputData()`. However, this is input that is created outside the processing of any individual request; it is input that is defined when the chain is defined.

In addition, a `Worker` can provide *output* data to factory methods like `success()` on `ListenableWorker.Result`. Those factory methods take the same sort of `Data` object that `setInputData()` does. The output data can be used in two places:

- If this request has another request chained after it, that later request receives the earlier request's output data as input.
- The output data is available from the `WorkInfo` once the work is finished, so consumers of the `LiveData` status stream can also see the output data.

OK, Where's the Code?

The UnZIPWork sample module in the [Sampler](#) and [SamplerJ](#) projects are a variation on the previous example, this time where we have two requests in a chain.

DownloadWorker is largely the same as before, with two differences:

1. Rather than receiving a filename as input, it decides what the filename will be, as that will merely serve as a temporary file
2. It passes the path to that file to the next request in the chain via `setOutputData()`

```
package com.commonware.jetpack.work.download;

import android.content.Context;
import android.util.Log;
import java.io.File;
import java.io.IOException;
import androidx.annotation.NonNull;
import androidx.work.Data;
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okio.BufferedSink;
import okio.Okio;

public class DownloadWorker extends Worker {
    public static final String KEY_URL="url";
    public static final String KEY_RESULTDIR="resultDir";

    public DownloadWorker(@NonNull Context context,
                          @NonNull WorkerParameters workerParams) {
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        OkHttpClient client=new OkHttpClient();
        Request request=new Request.Builder()
            .url(getInputData().getString(KEY_URL))
            .build();
```

WORKING WITH WORKMANAGER

```
File dir=getApplicationContext().getCacheDir();
File downloadedFile=new File(dir, "temp.zip");

if (downloadedFile.exists()) {
    downloadedFile.delete();
}

try (Response response=client.newCall(request).execute()) {
    BufferedSink sink=Okio.buffer(Okio.sink(downloadedFile));

    sink.writeAll(response.body().source());
    sink.close();
}
catch (IOException e) {
    Log.e(getClass().getSimpleName(), "Exception downloading file", e);

    return ListenableWorker.Result.failure();
}

return ListenableWorker.Result.success(new Data.Builder()
    .putString(UnZIPWorker.KEY_ZIPFILE, downloadedFile.getAbsolutePath())
    .build());
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadWorker.java](#))

```
package com.commonsware.jetpack.work.download

import android.content.Context
import android.util.Log
import androidx.work.Data
import androidx.work.Worker
import androidx.work.WorkerParameters
import okhttp3.OkHttpClient
import okhttp3.Request
import okio.buffer
import okio.sink
import java.io.File
import java.io.IOException

class DownloadWorker(context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams) {

    override fun doWork(): Result {
        val client = OkHttpClient()
        val request = Request.Builder()
            .url(inputData.getString(KEY_URL)!!)
    }
}
```

```
.build()

val dir = applicationContext.cacheDir
val downloadedFile = File(dir, "temp.zip")

if (downloadedFile.exists()) {
    downloadedFile.delete()
}

try {
    client.newCall(request).execute().use { response ->
        val sink = downloadedFile.sink().buffer()

        response.body?.let { sink.writeAll(it.source()) }
        sink.close()
    }
} catch (e: IOException) {
    Log.e(javaClass.simpleName, "Exception downloading file", e)

    return Result.failure()
}

return Result.success(
    Data.Builder()
        .putString(UnZIPWorker.KEY_ZIPFILE, downloadedFile.absolutePath)
        .build()
)
}

companion object {
    const val KEY_URL = "url"
    const val KEY_RESULTDIR = "resultDir"
}
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadWorker.kt](#))

We now also have an UnZIPWorker. This expects two pieces of input: the file to unZIP and the directory to unZIP it into. It uses [the CWAC-Security library](#) and its `ZipUtils.unzip()` method, as that safely handles possibly-malicious ZIP files (e.g., zip bombs):

```
package com.commonsware.jetpack.work.download;

import android.content.Context;
import android.util.Log;
import com.commonsware.cwac.security.ZipUtils;
import java.io.File;
import androidx.annotation.NonNull;
```

WORKING WITH WORKMANAGER

```
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class UnZIPWorker extends Worker {
    public static final String KEY_ZIPFILE="zipFile";
    public static final String KEY_RESULTDIR="resultDir";

    public UnZIPWorker(@NonNull Context context,
                      @NonNull WorkerParameters workerParams) {
        super(context, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        File downloadedFile=new File(getInputData().getString(KEY_ZIPFILE));
        File dir=getApplicationContext().getCacheDir();
        String resultDirData=getInputData().getString(KEY_RESULTDIR);
        File resultDir=new File(dir, resultDirData==null ? "results" : resultDirData);

        try {
            ZipUtils.unzip(downloadedFile, resultDir, 2048, 1024*1024*16);
            downloadedFile.delete();
        }
        catch (Exception e) {
            Log.e(getClass().getSimpleName(), "Exception unzipping file", e);

            return ListenableWorker.Result.failure();
        }

        return ListenableWorker.Result.success();
    }
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/UnZIPWorker.java](#))

```
package com.commonsware.jetpack.work.download

import android.content.Context
import android.util.Log
import androidx.work.ListenableWorker
import androidx.work.Worker
import androidx.work.WorkerParameters
import com.commonsware.cwac.security.ZipUtils
import java.io.File

class UnZIPWorker(context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams) {

    override fun doWork(): Result {
        val downloadedFile = File(inputData.getString(KEY_ZIPFILE)!!)
        val dir = applicationContext.cacheDir
        val resultDirData = inputData.getString(KEY_RESULTDIR)
        val resultDir = File(dir, resultDirData ?: "results")
    }
}
```

```
try {
    ZipUtils.unzip(downloadedFile, resultDir, 2048, 1024 * 1024 * 16)
    downloadedFile.delete()
} catch (e: Exception) {
    Log.e(javaClass.simpleName, "Exception unZIPing file", e)

    return Result.failure()
}

return Result.success()
}

companion object {
    const val KEY_ZIPFILE = "zipFile"
    const val KEY_RESULTDIR = "resultDir"
}
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/UnZIPWorker.kt](#))

DownloadViewModel now sets up a request chain using both worker classes:

```
package com.commonsware.jetpack.work.download;

import android.app.Application;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

public class DownloadViewModel extends AndroidViewModel {
    public final MediatorLiveData<WorkInfo> liveWorkStatus=new MediatorLiveData<>();

    public DownloadViewModel(@NonNull Application application) {
        super(application);
    }

    public void doTheDownload() {
        OneTimeWorkRequest downloadWork=
            new OneTimeWorkRequest.Builder(DownloadWorker.class)
                .setConstraints(new Constraints.Builder()
                    .setRequiredNetworkType(NetworkType.CONNECTED)
                    .setRequiresBatteryNotLow(true)
                    .build())
                .setInputData(new Data.Builder()
                    .putString(DownloadWorker.KEY_URL,
                        "https://commonsware.com/Android/source_1_0.zip")
```

WORKING WITH WORKMANAGER

```
        .build()
        .addTag("download")
        .build();
OneTimeWorkRequest unZIPWork=
    new OneTimeWorkRequest.Builder(UnZIPWorker.class)
        .setConstraints(new Constraints.Builder()
            .setRequiresStorageNotLow(true)
            .setRequiresBatteryNotLow(true)
            .build())
        .setInputData(new Data.Builder()
            .putString(DownloadWorker.KEY_RESULTDIR, "unzipped")
            .build())
        .addTag("unZIP")
        .build();

WorkManager.getInstance(getApplication())
    .beginWith(downloadWork)
    .then(unZIPWork)
    .enqueue();

final LiveData<WorkInfo> liveOpStatus=
    WorkManager.getInstance(getApplication()).getWorkInfoByIdLiveData(unZIPWork.getId());

liveWorkStatus.addSource(liveOpStatus, workStatus -> {
    liveWorkStatus.setValue(workStatus);

    if (workStatus.getState().isFinished()) {
        liveWorkStatus.removeSource(liveOpStatus);
    }
});
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
package com.commonsware.jetpack.work.download

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MediatorLiveData
import androidx.work.Constraints
import androidx.work.Data
import androidx.work.NetworkType
import androidx.work.OneTimeWorkRequest
import androidx.work.WorkInfo
import androidx.work.WorkManager

class DownloadViewModel(application: Application) :
    AndroidViewModel(application) {
    val liveWorkStatus = MediatorLiveData<WorkInfo>()

    fun doTheDownload() {
        val downloadWork = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
            .setConstraints(
```

```
        Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresBatteryNotLow(true)
            .build()
    )
    .setInputData(
        Data.Builder()
            .putString(
                DownloadWorker.KEY_URL,
                "https://commonsware.com/Android/source_1_0.zip"
            )
            .build()
    )
    .addTag("download")
    .build()
val unZIPWork = OneTimeWorkRequest.Builder(UnZIPWorker::class.java)
    .setConstraints(
        Constraints.Builder()
            .setRequiresStorageNotLow(true)
            .setRequiresBatteryNotLow(true)
            .build()
    )
    .setInputData(
        Data.Builder()
            .putString(DownloadWorker.KEY_RESULTDIR, "unzipped")
            .build()
    )
    .addTag("unZIP")
    .build()

WorkManager.getInstance(getApplication())
    .beginWith(downloadWork)
    .then(unZIPWork)
    .enqueue()

val liveOpStatus = WorkManager.getInstance(getApplication())
    .getWorkInfoByIdLiveData(unZIPWork.id)

liveWorkStatus.addSource(liveOpStatus) { workStatus ->
    liveWorkStatus.value = workStatus

    if (workStatus.state.isFinished) {
        liveWorkStatus.removeSource(liveOpStatus)
    }
}
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

Of note:

- downloadWork is defined the same as before, except that we skip supplying the filename, and the URL now points to a ZIP file instead of a PDF
- unZIPWork does not require an Internet connection, but it does require that we have a reasonable amount of storage available
- unZIPWork gets the name of a directory to create in `getCacheDir()` to hold the unZIPped results
- We use `beginWith()` and `then()` to set up the chain, using `enqueue()` to enqueue the results
- We monitor the unZIPWork status for the purposes of re-enabling the button and showing the Toast

In principle, we should be monitoring *both* requests' status updates. If the first request fails for some reason (e.g., HTTP 404 error), the second request will never run. We could do that by calling `getWorkInfosLiveData()` on the `WorkContinuation`, which returns a `LiveData` of a *list* of `WorkInfo` objects, one for each request in the chain. That significantly increases the complexity of the sample (e.g., what do we do for data binding in this case?), and so we cheat for the sake of brevity.

How Complex Can This Get?

It can get as complicated as you like:

- You can keep chaining work together by successive `then()` calls:

```
WorkManager.getInstance(getApplicationContext())
    .beginWith(lets)
    .then(go)
    .then(crazy)
    .enqueue();
```

- You can have parallel requests as part of a chain, by passing multiple `WorkRequest` objects to `beginWith()` or `then()`
- You can chain a `WorkContinuation` onto another `WorkContinuation`
- You can create `InputMerger` implementations to help coordinate out the output data from previous steps in the chain are merged together to form the input data for successive steps in the chain
- And so on

However, while WorkManager is useful for deferrable tasks, it is not a full workflow system:

- There is limited ability to cancel work, as noted previously
- There is no ability to change enqueued work, except by trying to cancel it and then enqueueing its replacement
- There are no specifications for how long any individual request or an entire chain can take, in terms of time
- There are no specifications for how results are handled when it takes multiple process invocations to complete a chain (e.g., a long chain extending past the 10-minute limit)
- And so on

As a result, at least for the time being, be careful when trying to create complex WorkRequest chains.

Periodic Work

So far, all of the work has been for single tasks, using `OneTimeWorkRequest`. The other WorkRequest implementation is `PeriodicWorkRequest`, and as the name suggests, it is for work that should repeat with a given interval.

The interval is provided via the `PeriodicWorkRequest.Builder` constructor, either as a `Duration` or as a long and `TimeUnit` pair. There is a minimum allowed period, defined as `PeriodicWorkRequest.MIN_PERIODIC_INTERVAL_MILLIS`, presently defined as 15 minutes. Consider the supplied interval to be a suggestion, rather than a requirement.

Unique Work

Sometimes, we want to avoid accidentally enqueueing the same work more than once.

For example, suppose that the app has a `Worker` that pulls data from a server. Normally, that is triggered by a push message (e.g., Firebase Cloud Messaging). However, you also have it set up that user actions can trigger that work to be done, such as via a manual refresh option (e.g., pull-to-refresh). What you want to avoid is trying to do two of this bit of work simultaneously, as that might confuse things on either the device side or the server side.

To handle this, instead of `enqueue()` on `WorkManager`, you can use `beginUniqueWork()`. This takes a “name” that identifies this logical unit of work. If you previously used `beginUniqueWork()`, and you later call `beginUniqueWork()` with the same name, and the earlier work is still ongoing, you can specify what should happen (e.g., ignore the new work request).

Note that this does not support periodic work: you can only coordinate `OneTimeWorkRequest`, not `PeriodicWorkRequest`.

Testing Work

The work-testing artifact offers a `WorkManagerTestInitHelper` utility class to help with instrumented testing.

First, it has `initializeTestWorkManager()`. This configures `WorkManager` to use a `SynchronousExecutor`. This amounts to a mock `Executor`, one that runs supplied `Runnable` objects immediately on the current thread. By using `SynchronousExecutor`, your `enqueue()` calls for `WorkManager` will happen immediately and synchronously, rather than asynchronously.

Also, `WorkManagerTestInitHelper` has a `getTestDriver()` method, which returns a `TestDriver`. This offers a `setAllConstraintsMet()` method, which takes a work request ID and tells `WorkManager` that all of the constraints for that work request are met. This makes your tests more deterministic, since constraints are normally there to test the environment, and that might change from run to run of your tests. However, it is very important to call `setAllConstraintsMet()` **after** you `enqueue()` the work:

```
WorkManager.getInstance(context).enqueue(work);
WorkManagerTestInitHelper.getTestDriver(context).setAllConstraintsMet(work.getId());
```

Note: calling `setAllConstraintsMet()` before calling `enqueue()` results in a crash.

The Work/Download sample app contains a `DownloadWorkerTest` class that shows the use of `WorkManagerTestInitHelper` and `TestDriver`:

```
package com.commonware.jetpack.work.download;

import android.content.Context;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
import java.io.File;
import androidx.test.ext.junit.runners.AndroidJUnit4;
import androidx.test.platform.app.InstrumentationRegistry;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkManager;
import androidx.work.WorkRequest;
import androidx.work.testing.WorkManagerTestInitHelper;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

@RunWith(AndroidJUnit4.class)
public class DownloadWorkerTest {
    private File expected;
    private final Context context =
        InstrumentationRegistry.getInstrumentation().getTargetContext();

    @Before
    public void setUp() {
        WorkManagerTestInitHelper.initializeTestWorkManager(context);

        expected = new File(context.getCacheDir(), "oldbook.pdf");

        if (expected.exists()) {
            expected.delete();
        }
    }

    @Test
    public void download() {
        assertFalse(expected.exists());

        WorkManager.getInstance(context).enqueue(buildWorkRequest(null));

        assertTrue(expected.exists());
    }

    @Test
    public void downloadWithConstraints() {
        Constraints constraints = new Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresBatteryNotLow(true)
            .build();
        WorkRequest work = buildWorkRequest(constraints);

        assertFalse(expected.exists());
    }
}
```

```
WorkManager.getInstance(context).enqueue(work);
WorkManagerTestInitHelper.getTestDriver(context)
    .setAllConstraintsMet(work.getId());

assertTrue(expected.exists());
}

private WorkRequest buildWorkRequest(Constraints constraints) {
    OneTimeWorkRequest.Builder builder =
        new OneTimeWorkRequest.Builder(DownloadWorker.class)
            .setInputData(new Data.Builder()
                .putString(DownloadWorker.KEY_URL,
                    "https://commonsware.com/Android/Android-1_0-CC.pdf")
                .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
                .build())
            .addTag("download");

    if (constraints != null) {
        builder.setConstraints(constraints);
    }

    return builder.build();
}
```

(from [DownloadWork/src/androidTest/java/com/commonsware/jetpack/work/download/DownloadWorkerTest.java](#))

```
package com.commonsware.jetpack.work.download

import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import androidx.work.*
import androidx.work.testing.WorkManagerTestInitHelper
import org.junit.Assert.assertFalse
import org.junit.Assert.assertTrue
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import java.io.File

@RunWith(AndroidJUnit4::class)
class DownloadWorkerTest {
    private lateinit var expected: File
    private val context =
        InstrumentationRegistry.getInstrumentation().targetContext

    @Before
    fun setUp() {
        WorkManagerTestInitHelper.initializeTestWorkManager(context)

        expected = File(context.cacheDir, "oldbook.pdf")
    }
}
```

```
        if (expected.exists()) {
            expected.delete()
        }
    }

    @Test
    fun download() {
        assertFalse(expected.exists())

        WorkManager.getInstance(context).enqueue(buildWorkRequest(null))

        assertTrue(expected.exists())
    }

    @Test
    fun downloadWithConstraints() {
        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresBatteryNotLow(true)
            .build()
        val work = buildWorkRequest(constraints)

        assertFalse(expected.exists())

        WorkManager.getInstance(context).enqueue(work)
        WorkManagerTestInitHelper.getTestDriver(context)!!.setAllConstraintsMet(work.id)

        assertTrue(expected.exists())
    }

    private fun buildWorkRequest(constraints: Constraints?): WorkRequest {
        val builder = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
            .setInputData(
                Data.Builder()
                    .putString(
                        DownloadWorker.KEY_URL,
                        "https://commonsware.com/Android/Android-1_0-CC.pdf"
                    )
                    .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
                    .build()
            )
            .addTag("download")

        if (constraints != null) {
            builder.setConstraints(constraints)
        }

        return builder.build()
    }
}
```

(from [DownloadWork/src/androidTest/java/com/commonsware/jetpack/work/download/DownloadWorkerTest.kt](#))

This class tests `DownloadWorker` both with and without constraints, validating that the output file exists after the work has been done. Since we are using the synchronous test configuration of `WorkManager`, we can test this work without having to resort to `CountDownLatch` or similar tricks for testing multithreaded code.

Independent of work-testing, note that Worker has some dependencies on Context, and it may be difficult to mock that Context since you are not the one providing it. It may be necessary to consider your Worker as something to be tested with instrumented tests, as we are doing here. If you wish to have deferred tasks be unit tested outside of Android, consider isolating that logic in another class that your Worker then happens to use.

Inspecting Work

Android Studio Arctic Fox expanded the App Inspection tool beyond the original [Database Inspector](#) for Room. It now contains a “Background Task Inspector” that lets you know what WorkManager work has been scheduled or has completed, along with the details of that work.

In addition to requiring Arctic Fox (or newer), this tool also requires that your app use 2.5.0 or higher of the WorkManager dependencies.

Starting the Inspector

As with the Database Inspector, you get to the Background Task Inspector via the “App Inspection” tool window. If this is not already docked on the edges of your IDE window, you can open it from View > Tool Windows > App Inspection from the main menu.

You also need your app that is using WorkManager to be running in order to get to the Background Task Inspector tab. You can run the app first, then open the App Inspection tool window, or open the tool window first and then run the app. If your process is not selected automatically, a menu in the tool window will let you choose it.

Examining the Results

The Background Task Inspector tool window has several sub-views that give you details of the pending, in-progress, and completed pieces of work.

Note that WorkManager stores its work information in a Room database, and so the details of completed pieces of work persist past process termination. It is not quite clear when those entries get pruned from the database.

The Table

The default view is a table showing the work. Initially, that table will be empty, but if you schedule some WorkManager work, the table will show that work:

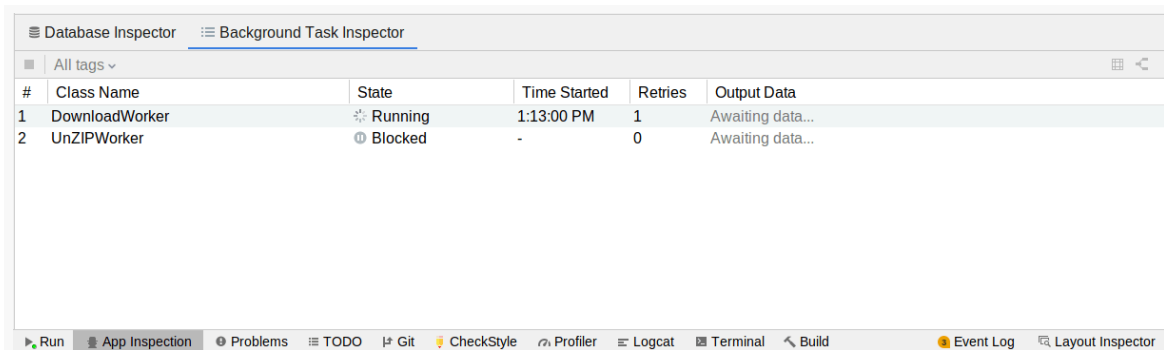


Figure 235: Background Task Inspector, Showing In-Progress and Future Work

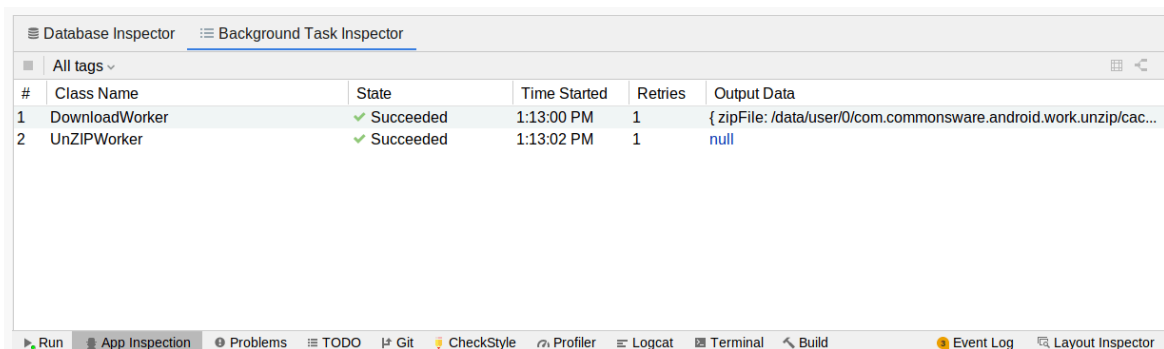


Figure 236: Background Task Inspector, Completed Work

You are given:

- The Java/Kotlin class that implements the Worker interface that is handling the work
- What state that work is in (e.g., running, blocked, succeeded, failed)
- When the work entered the running state
- Whether the work was retried due to some problem (1 means it was never retried)
- A quasi-JSON depiction of what output data was supplied by the work (or null if the work had no output data)

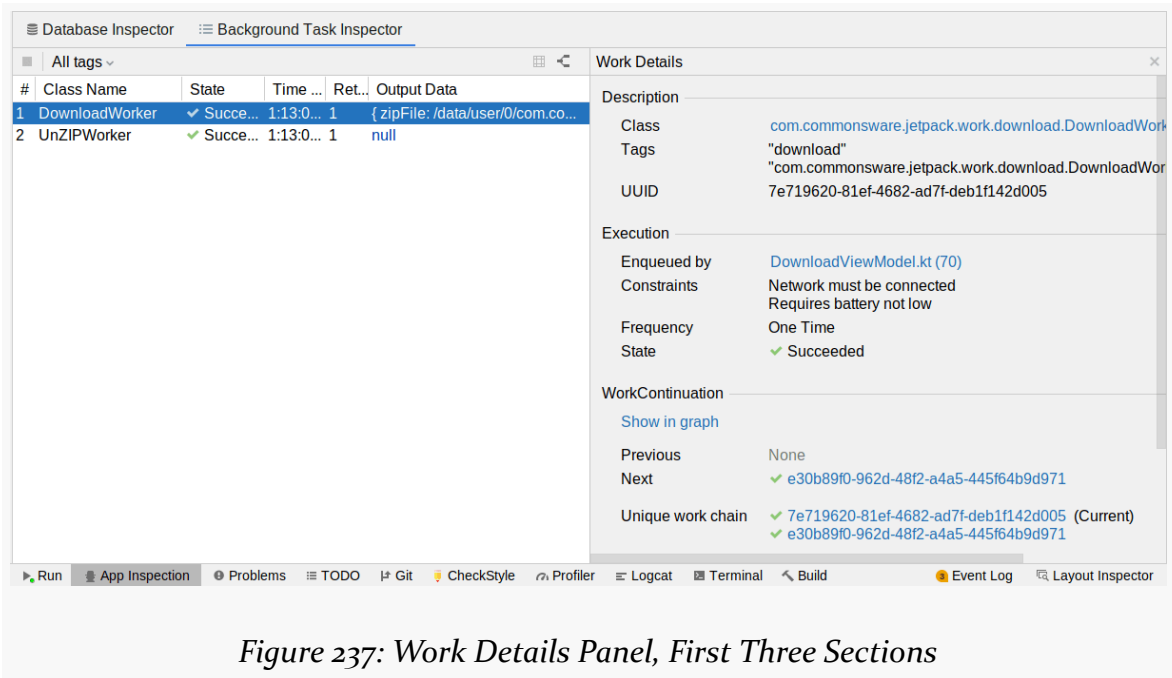
In these screenshots, we have the work from the UnZIPWork sample app. The first screenshot shows the in-progress download work, while the second screenshot

WORKING WITH WORKMANAGER

shows the results of the completed chain of work.

The Work Details Panel

If you click on a row in the table, a side panel will open with details of the particular piece of work that you chose:



WORKING WITH WORKMANAGER

The panel is long and probably will require you to scroll to see everything:

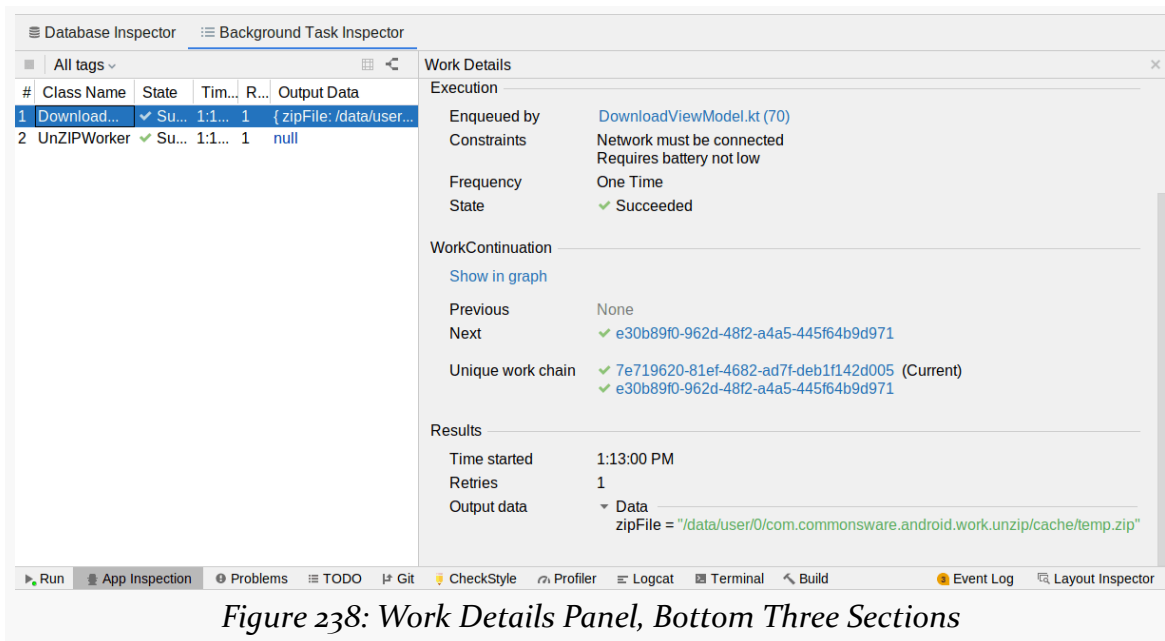


Figure 238: Work Details Panel, Bottom Three Sections

Here you can find out everything that WorkManager knows about that piece of work, including:

- The generated UUID that uniquely identifies the work
- What code requested the work
- What the constraints were on when the work should be done
- Whether this was a one-time piece of work or an instance of a periodic work request
- The UUIDs of any preceding or follow-on pieces of work in a chain

The Flow Diagram

For simple chains of work, there is a good chance that the table is all that you will need to determine what happened with your requested bits of work. But if you have a complex work chain, with branching and merging and so forth, there is a flowchart-style icon in the top bar of the table that switches that view to a flowchart-style representation of the work chain:

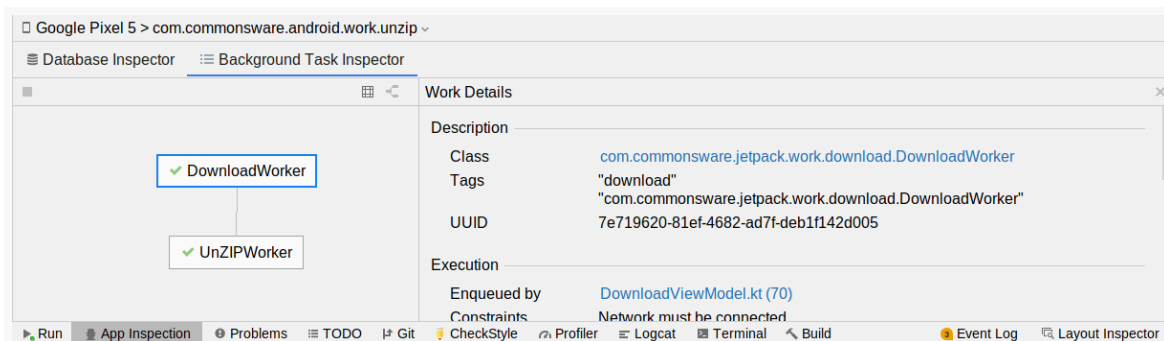


Figure 239: Background Task Inspector Flowcart View, With a Trivial Flowchart

The icon next to the flowchart one switches you back to the table view.

The Tag Filter

You might have a lot of work.

To help you sift through it, you can filter the table view. There is a drop-down in the table's top bar that defaults to "All tags" — clicking it gives you options of filtering by Java/Kotlin class or by actual tags that you attached to your work:

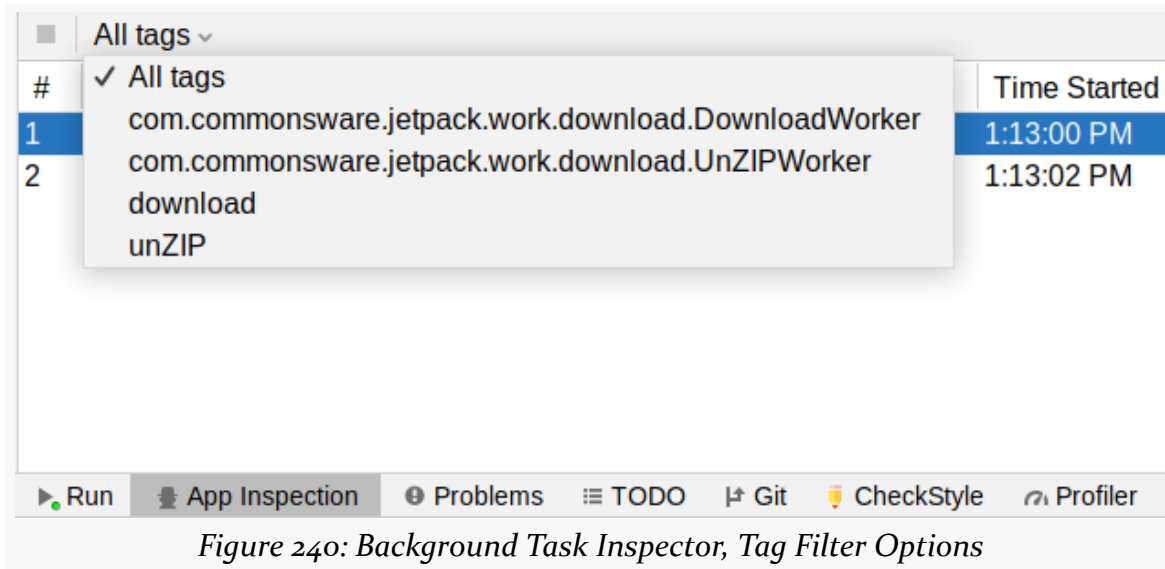


Figure 240: Background Task Inspector, Tag Filter Options

WorkManager and Side Effects

WorkManager has a fairly clean and easy API and hides a lot of the complexity of scheduling background work that is not time-sensitive.

However, it has side effects.

To be able to restart your scheduled work after a reboot, WorkManager registers an `ACTION_BOOT_COMPLETED` receiver named `androidx.work.impl.background.systemalarm.RescheduleReceiver`. To be a good citizen, WorkManager only enables that receiver when you have relevant work and disables it otherwise. That way, your app does not unnecessarily slow down the boot process if there is no reason for your app to get control at boot time.

However, enabling and disabling a component, such as a receiver, triggers an `ACTION_PACKAGE_CHANGED` broadcast. Few apps directly have any code that watches for this broadcast, let alone would be harmed by having that broadcast be sent more times than might otherwise be necessary.

App widgets, though, *are* affected by `ACTION_PACKAGE_CHANGED`. Specifically,

`ACTION_PACKAGE_CHANGED` triggers an `onUpdate()` call to your `AppWidgetProvider`.

That too may not be a problem for most app widgets. Ideally, your `AppWidgetProvider` makes no assumptions about when, or how frequently, it gets called with `onUpdate()`. However, there is one area where this *is* a problem: with an `AppWidgetProvider` scheduling work in `onUpdate()`. The flow then becomes:

- A “regular” `onUpdate()` call comes in
- Your `AppWidgetProvider` schedules some work with `WorkManager`
- `WorkManager` enables `RescheduleReceiver`
- That triggers `ACTION_PACKAGE_CHANGED`, which triggers an `onUpdate()` call
- Your `AppWidgetProvider` schedules some work with `WorkManager` again
- `WorkManager` eventually gets through those two pieces of work
- `WorkManager` disables `RescheduleReceiver`, since it is no longer needed
- That triggers `ACTION_PACKAGE_CHANGED`, which triggers an `onUpdate()` call
- Your `AppWidgetProvider` schedules some work with `WorkManager`
- `WorkManager` enables `RescheduleReceiver`
- And we’re in an infinite loop

[The recommendation from Google](#) is to avoid unconditionally scheduling work with `WorkManager` from `onUpdate()`. Instead, only do it if you know that the work is needed and that it is safe to do so, meaning that you will not get into the infinite loop.

That advice may be difficult for some to implement.

This, and any other possible side-effects of `WorkManager`, are not documented. So, you need to be a bit careful about your use of `WorkManager`:

- If your app responds to `ACTION_PACKAGE_CHANGED` broadcasts, directly or indirectly, it may not be safe to schedule work there, lest you wind up in the infinite loop scenario described above.
- If your app responds to `ACTION_BATTERY_OK`, `ACTION_BATTERY_LOW`, `ACTION_POWER_CONNECTED`, `ACTION_POWER_DISCONNECTED`, `ACTION_DEVICE_STORAGE_LOW`, `ACTION_DEVICE_STORAGE_OK`, `CONNECTIVITY_CHANGE`, `ACTION_TIME_SET`, or `ACTION_TIMEZONE_CHANGED`, bear in mind that `WorkManager` has receivers for those broadcasts in your app. These are all disabled at the outset, but presumably `WorkManager` has code to enable them based on certain conditions, such as certain constraints that you set in your work requests. Be careful about scheduling work with `WorkManager` on those broadcasts as well.

- If your app responds to ACTION_BOOT_COMPLETED broadcasts, bear in mind that WorkManager also depends on this broadcast. Your respective receivers might be invoked in any order. It may not be safe to schedule work here, as WorkManager might assume that its own ACTION_BOOT_COMPLETED receiver has completed its work by the time you try scheduling new work. While I would not expect an infinite loop scenario, this is the sort of edge case that requires a lot of testing to ensure everything will work as expected.
- WorkManager has a ContentProvider that it bakes into your app as well. While scheduling your own work from onCreate() of a ContentProvider would be rather odd, it's possible that somebody might want to do that. Be careful, as WorkManager may not be fully ready for operation at that point. Note that, last time I tested it, all ContentProvider instances are created before onCreate() of Application, so *probably* it is safe to schedule work there.

There may be other edge and corner cases beyond these. So, while WorkManager is nice, make sure that you thoroughly test your use of it.

Part Four: Other Notable Topics

Creating a New Project

Most of the time, you will be doing Android development on an existing project, such as one created by somebody else on your development team.

But, on occasion, you will want to start development of a brand-new project. You have two main options for accomplishing that:

1. Clone some existing project. An Android app project is just a directory of files, so you can make a copy of an existing app directory and modify the copy to serve as the base for your new project. We will examine that scenario [later in this chapter](#).
2. Use the Android Studio new-project wizard to create a brand new project. This will give you a project that resembles the HelloWorld one seen in the early chapters of this book. We will examine that scenario [shortly](#).

Key Decisions That You Need to Make

Regardless of how you intend to start your new project, you will have a few decisions that you will need to make towards the outset.

Application ID

The most important one is the application ID. This is the unique identifier for your app:

- Two apps cannot be installed on the same device at the same time with the same application ID
- Two apps cannot be distributed through the Play Store at the same time with the same application ID (and other distribution channels may have similar

limitations)

From a technical standpoint, an application ID needs to be a valid Java/Kotlin package name (e.g., `com.commonware.this.is.valid`). Beyond that, though, you want to try to avoid any accidental collisions with the application IDs of other developers.

The recommended approach is to:

- Purchase a domain name, perhaps one that you intend to use for marketing the app
- Reverse the domain name (e.g., if your domain is `thisismydomainname.com`, the reversed domain name is `com.thisismydomainname`)
- Append some segment to it that identifies your app (e.g., `com.thisismydomainname.superapp`)
- Use that as your application ID

The reverse domain name convention reduces the odds of accidental collisions with other developers. The app-identifying segment at the end reduces the odds of accidental collisions with other developers in your organization. Even if you are a solo developer, you might create other Android apps in the future, so having an application ID that is tied to a specific app, rather than just a domain name, is a good idea.

Unlike the other decisions in this section, this one will be somewhat difficult to change. In particular, once you start shipping the app, you cannot change the application ID. If you do, that will be considered a totally different app by Android and app distribution channels (e.g., the Play Store).

Project Directory

Your project files need to live somewhere!

Language

You can have a project that:

- Is Java-only
- Allows Kotlin

(there is no simple way to have a Kotlin-only project)

Switching from a Java-only to a Java-and-Kotlin project is not that difficult. However, for future-proofing, the best choice is to support Kotlin from the outset, even if you do not plan on using it for much right away.

Minimum SDK Version

The `minSdkVersion` indicates how old of an Android version you are willing to support.

There are competing pressures here:

- The higher the `minSdkVersion`, the simpler your testing becomes, because you do not need to worry about as many Android OS versions
- Also, the higher the `minSdkVersion`, the simpler some aspects of development become, because you do not have to do quite as many different things for older versus newer devices
- The lower the `minSdkVersion`, the more prospective users there will be for your app, since not everybody has an Android device with an up-to-date version of Android

From a practical standpoint, a `minSdkVersion` below 14 will be exceptionally difficult, as that is the minimum supported API level for the Jetpack. Certain pieces of the Jetpack have higher minimums than that, though 14 is fairly common.

The New-Project Wizard

For many developers, the “go-to” approach for creating a new project is via the new-project wizard. Typically, you get to that wizard via “File” > “New” > “New Project...” from the main menu:

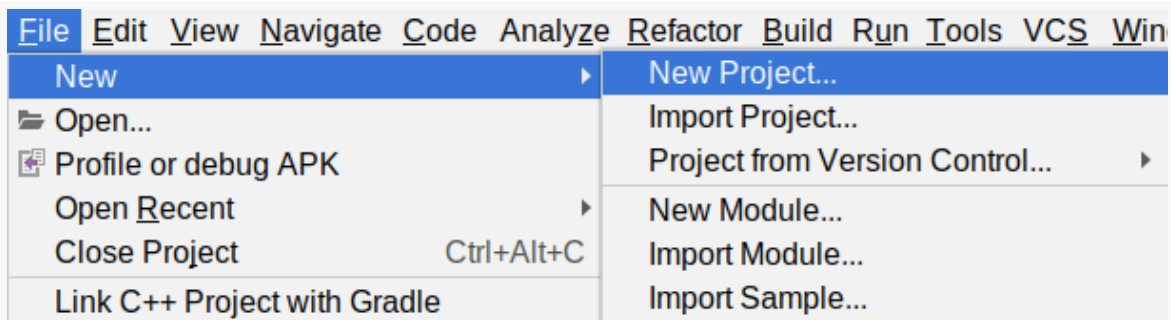
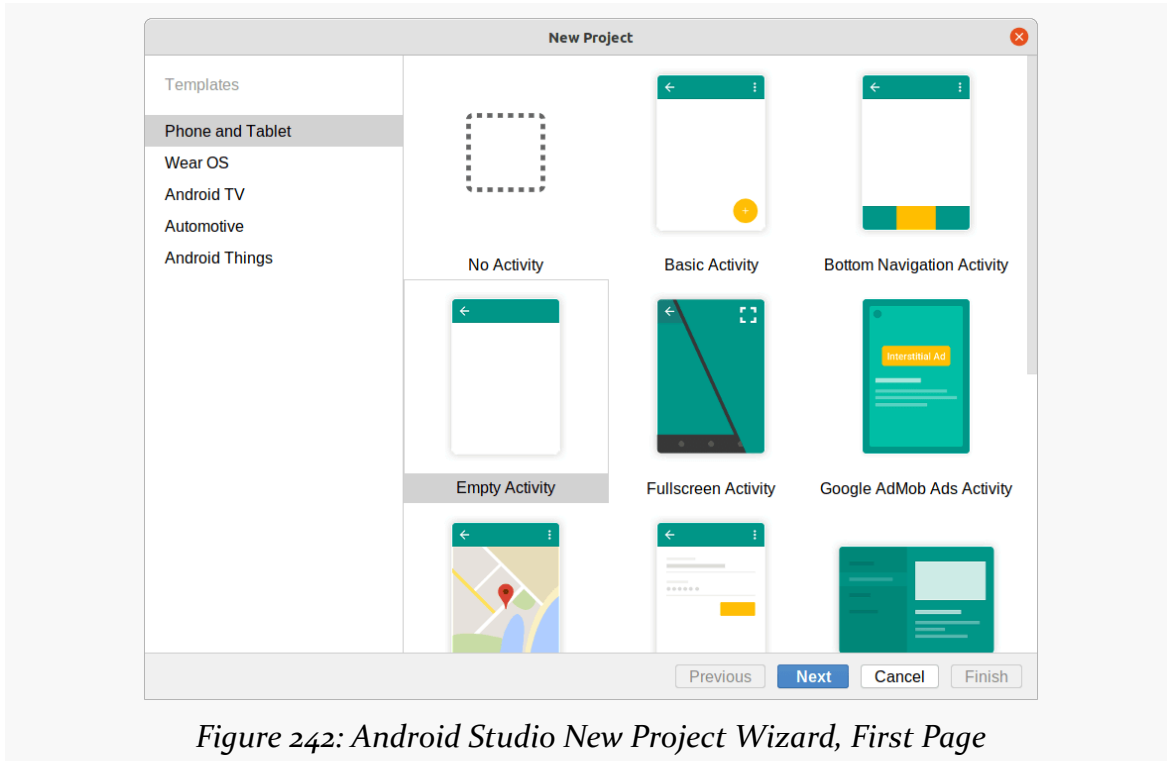


Figure 241: Android Studio New Project Main Menu Option

If you are at the “welcome dialog” and do not have a project open, there is a “Start a new Android Studio project” option that opens the new-project wizard.

Project Template

The first page of the wizard allows you to choose a template for your new project:



What You Get

The idea is that you would choose a template that matches the starting point that you want for the app. The new-project wizard will give you classes, resources, and so on that are based on the template. The theory is that you could then use those files as the starting point, tweaking them as needed and blending in your own application logic.

What Else You Need to Do

In reality, these templates are mostly for samples or scrap projects for experimentation. Invariably, you will replace or remove a substantial amount of what they create for you. As such, do not spend a lot of time worrying about which template to use for a new project. The “Empty Activity” template traditionally has the least amount of “cruft” to remove.

CREATING A NEW PROJECT

However, do not be afraid to play around with the templates in scrap projects. The output from the templates is not always the absolute best code, but it may help you learn a bit more about various features of Android, including some stuff not covered by this book.

Project Details

Once you choose a template and click the “Next” button, you will be taken to the second page of the new-project wizard, where you can provide the core details of the project that you want to create:

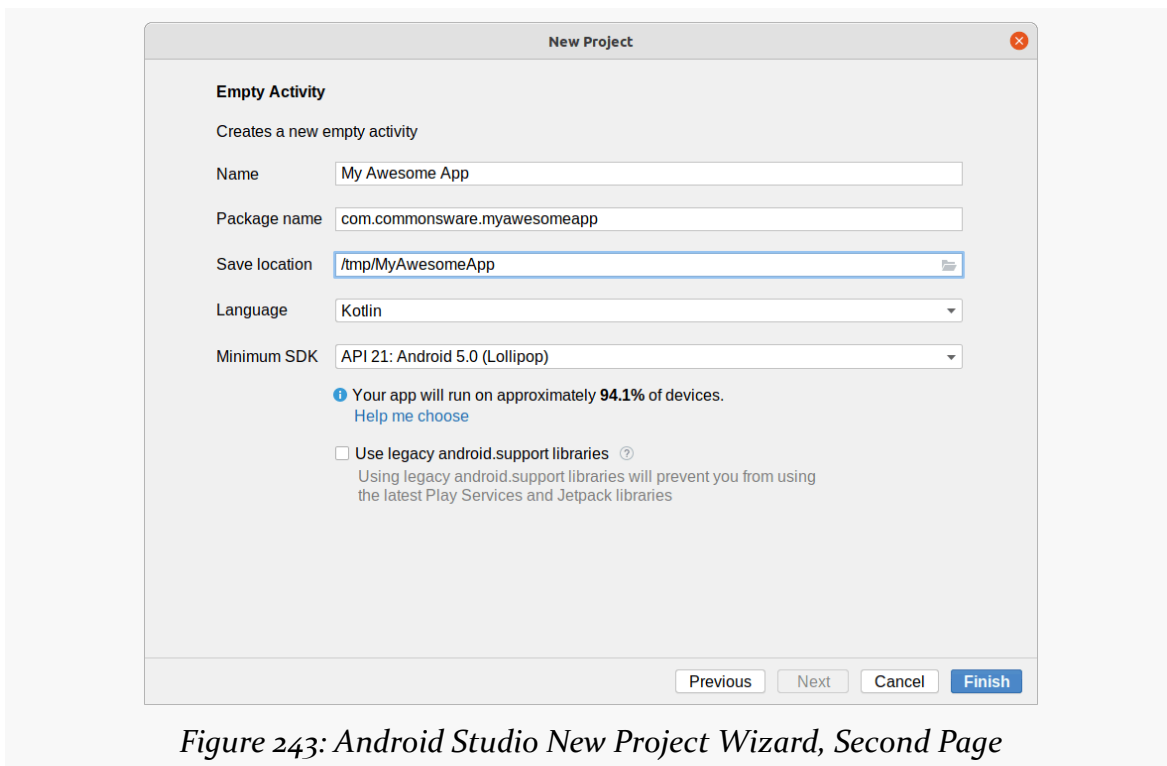


Figure 243: Android Studio New Project Wizard, Second Page

Core Elements

The “Name” field is the display name for your app. It will be placed in the `app_name` string resource and will be used in the manifest as the label for your app and activity.

The “package name” is your application ID. If you create several projects, Android Studio will remember the top-level domain that you use for your projects and will pre-fill that in as part of a generated sample application ID.

CREATING A NEW PROJECT

“Save location” is where on your development machine’s filesystem you would like your project to go. Android Studio will create the directory for you if it does not already exist, and Android Studio may complain if you try creating a project in a directory that exists and already has files in it.

Choosing the Minimum SDK Version

A drop-down lets you specify your desired `minSdkVersion` value. The “help me choose” link beneath it will bring up a dialog showing you highlights of what was added in each Android release, along with an approximate percentage of Google Play ecosystem devices run that API level or lower:

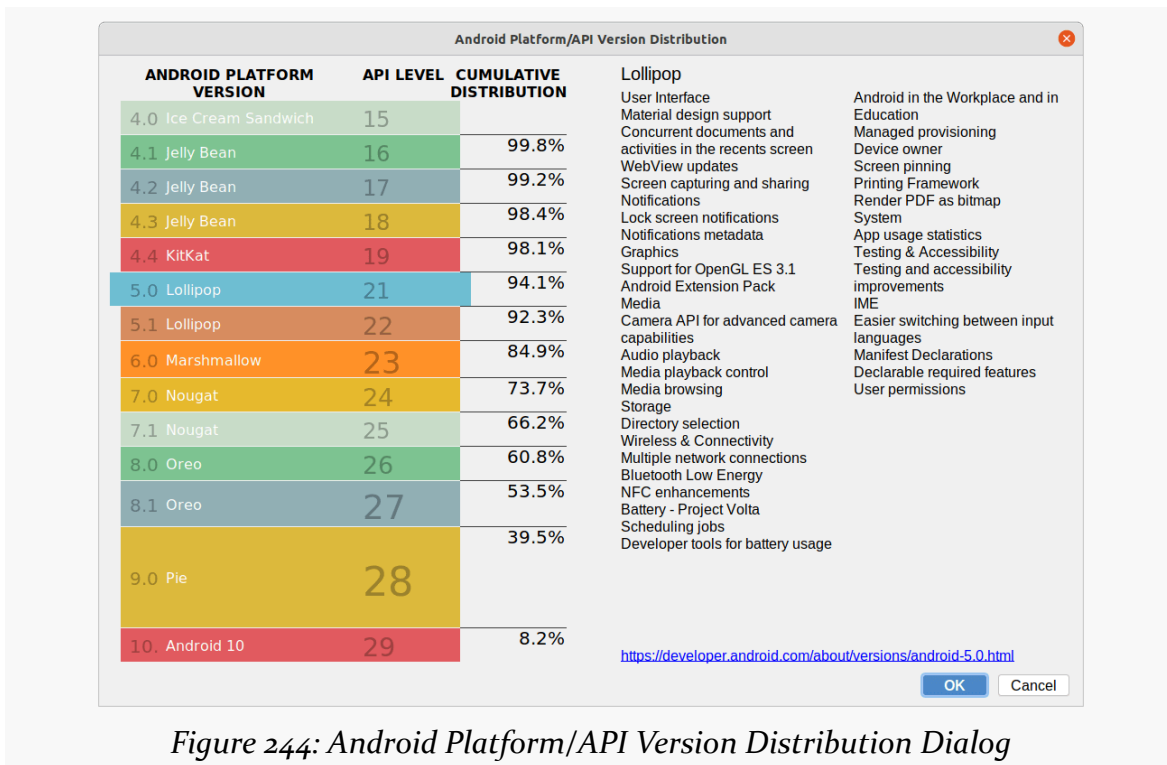


Figure 244: Android Platform/API Version Distribution Dialog

Additional Screens

Most of the templates do not ask for any additional information than what appears in the “Configure your project” wizard page. However, a few do continue to a third wizard page. The “Native C++” template is one:

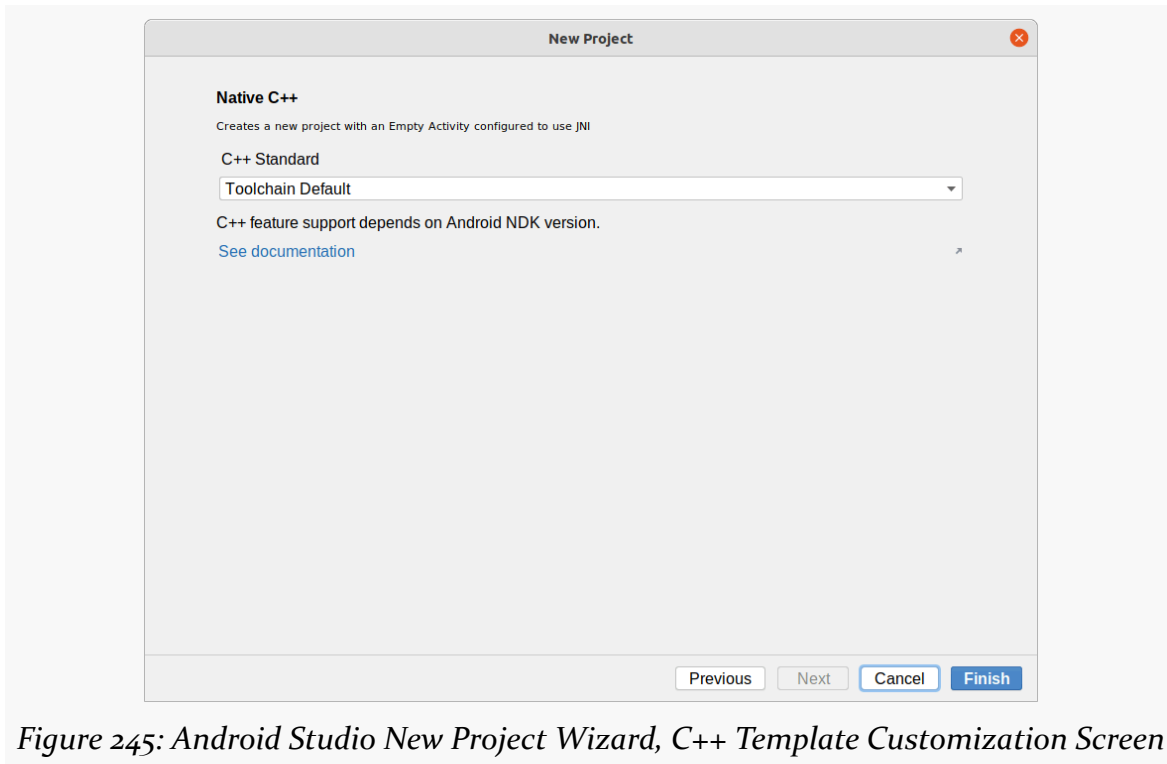


Figure 245: Android Studio New Project Wizard, C++ Template Customization Screen

This template sets up a project where all code is written in C/C++ instead of in Java/Kotlin. This third wizard page lets you choose what particular C++ version you want (C++11, C++14, or whatever the default it).

You’re Done!

Once you click the “Finish” button, Android Studio will create your project to your specifications. The standard IDE window will appear, showing you what it generated.

Copying an Existing Project

If you prefer, you can use an existing project as the starting point for creating a new project. An Android Studio project is just a directory tree of files. Copying portions

of that tree gives you a “new” project with all the existing content from the original one.

After copying the full tree, though, you may want to delete some existing stuff from the copy, before importing the copy into Android Studio. The items listed in the `.gitignore` file in the project root could be deleted. Notably, this usually means deleting:

- The `.gradle/` directory (note the leading `!`)
- The `.idea/` directory
- The `build/` directory in the project root and any modules
- The `*.iml` file in the project root and any modules

All of those will be rebuilt when you import the project into Android Studio, with proper values for this copy.

Also, if you are copying this directory from some other development machine, delete `local.properties` from the project root. This contains values that are unique for your development machine, and it typically varies in content between machines.

Of course, much (if not all) of the code and layouts in the copy are unnecessary, as they pertain to the original project, not the copy. Whether you bulk-delete this stuff or selectively delete or modify what is there is up to you. Also, you will need to edit things like the `applicationId` in `build.gradle` and the corresponding `package` attribute on the `<manifest>` element of `AndroidManifest.xml`.

If you find yourself doing this sort of thing a lot, you might consider setting up your own “template” project that you copy from, one that has the settings you want with the least stuff needing to be edited or deleted.

Creating a Library Module

Adding a library module to an existing project — whether the project was created using the new-project wizard or was imported from elsewhere — is fairly easy.

Using the Android Studio Wizard

From the Android Studio main menu, you can choose “File” > “New” > “New Module” to start a new-module wizard. Functionally, this works like the new-project wizard, though the UI is a bit different.

CREATING A NEW PROJECT

We start off with a list of templates on the left and details for the chosen template on the right:

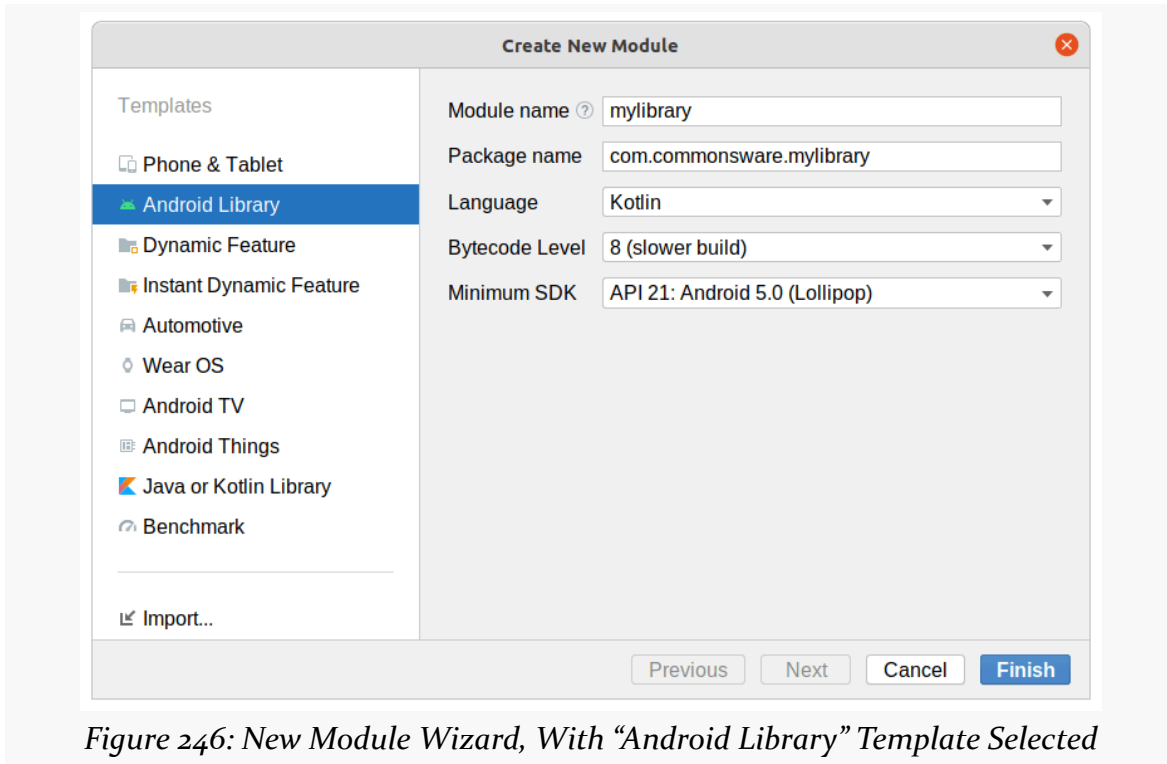


Figure 246: New Module Wizard, With “Android Library” Template Selected

The “Module name” field lets you give a name to the library. This will be the directory that is created off of the project root for the contents of the module. “Package name”, “Language”, and “Minimum SDK” are the same as their equivalents in the new-project wizard. “Bytecode Level” refers to the Java language versions supported by the library — you almost always want to choose the default value here.

Then, you can click “Finish”, and Android Studio will set up your new library module.

Reviewing What You Get

The exact details of what you get will vary by module template, language choice, and other settings in the new-module wizard. But, by and large, a library module will look a lot like your app module, in terms of general structure. It will have its own source sets, such as `main`, where you can put your Java/Kotlin code, your resources, your manifest, etc. It will have its own source sets for tests. And it will have a `build.gradle` file.

CREATING A NEW PROJECT

The big thing that distinguishes a library module from an application module is a choice of plugin.

In an application module, the module's `build.gradle` file will pull in the `com.android.application` module:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
}
```

In an Android library module, this will be `com.android.library` instead:

```
plugins {  
    id 'com.android.library'  
    id 'kotlin-android'  
}
```

This teaches Gradle that we are building a library, not an app.

In addition, there will be one more change: to the `settings.gradle` file in the project root directory. The primary purpose of this file is to list all of the modules in the project. Android Studio's new-module wizard helpfully adds an entry to `settings.gradle` for you:

```
rootProject.name = "My Application"  
include ':app'  
include ':mylibrary'
```

The leading-colon syntax says “this module is located directly off of the project root directory”.

Adding Your Own Stuff

From here, you can add your own code, resources, and anything else that you need, just like you would have in your app module. If you then have your app module depend upon the library (via `implementation project(":mylibrary")` or whatever), the things that you put into the library will be available in your app module. For example, the `Utilities` library module in the Kotlin sample project contains `ViewBindingFragment`, and other app modules depend upon `Utilities` to be able to reference that class.

Pondering `implementation` Versus `api`

There are some features that let you control what elements of the library module are accessible by modules that depend upon that library. Back in [the chapter on modules](#), we mentioned one: the use of the `internal` visibility keyword in Kotlin. `internal` things are visible within a library module but are treated as `private` by other modules that depend upon the library. You can decide what classes, functions, etc. are part of the library's API and mark the rest as `internal` or `private`.

A library can have its own dependencies, via the same sort of dependencies closure in `build.gradle` that an app module can have. However, akin to `public` versus `internal` in Kotlin, you have a choice for declaring dependencies in Gradle: `api` and `implementation`:

- `implementation` says “use this dependency in this module, but do not expose its contents to modules that depend upon this one”
- `api` says “use this dependency in this module, and also allow modules that depend upon this one to use its contents”

In other words: is this dependency itself part of your module's API (`api`), or is it part of your module's implementation (`implementation`). `api` is like having a public element in Kotlin: it is part of your API. `implementation` is like having an internal element in Kotlin: it is part of your module's implementation, but other modules should not get that specific dependency as a “transitive dependency”.

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.

Role of Code Signing

We digitally sign our apps to ensure that nobody tampers with those apps in ways that may harm the user.

App Updates

For example: you distribute your app with an application ID of `com.awesomecorp.fun`. What would stop somebody else from trying to distribute updates to that app, by shipping their own APK with the same application ID and a higher `versionCode`?

What prevents that is the digital signature. In order for an APK to be considered a valid upgrade for an installed app, it needs to:

- Have the same application ID
- Have a higher `versionCode`
- Be signed by the same signing key that signed the installed app

So long as nobody steals your signing key, and so long as there is no major breakthrough in falsifying digital signatures, only you can distribute updates to your app.

Tampering by Distributors

A related concern is whether your app distributor — Google, in the case of the Play Store — can modify your app before they distribute it.

If you sign your app, and you verify that the signature of the distributed app has not changed, then you know that the distributor did not change the app.

Conversely, if the *distributor* signs the app, then the distributor can do whatever it wants with the app. In effect, whoever signs the app really controls what the app is and does.

What Happens In Debug Mode

Of course, you may be wondering how you got this far in life without worrying about keys and signatures.

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 a problem mentioned 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:\Users\...\android\` on Windows

(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.

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 Android Studio 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
```

SIGNING YOUR APP

```
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. Here, we will focus on using Android Studio itself, though note that there are options for signing your app via Gradle tasks.

SIGNING YOUR APP

Start by opening up your project and going to “Build” > “Generate Signed Bundle/APK...” from the main menu. This brings up the first page of a signing wizard:

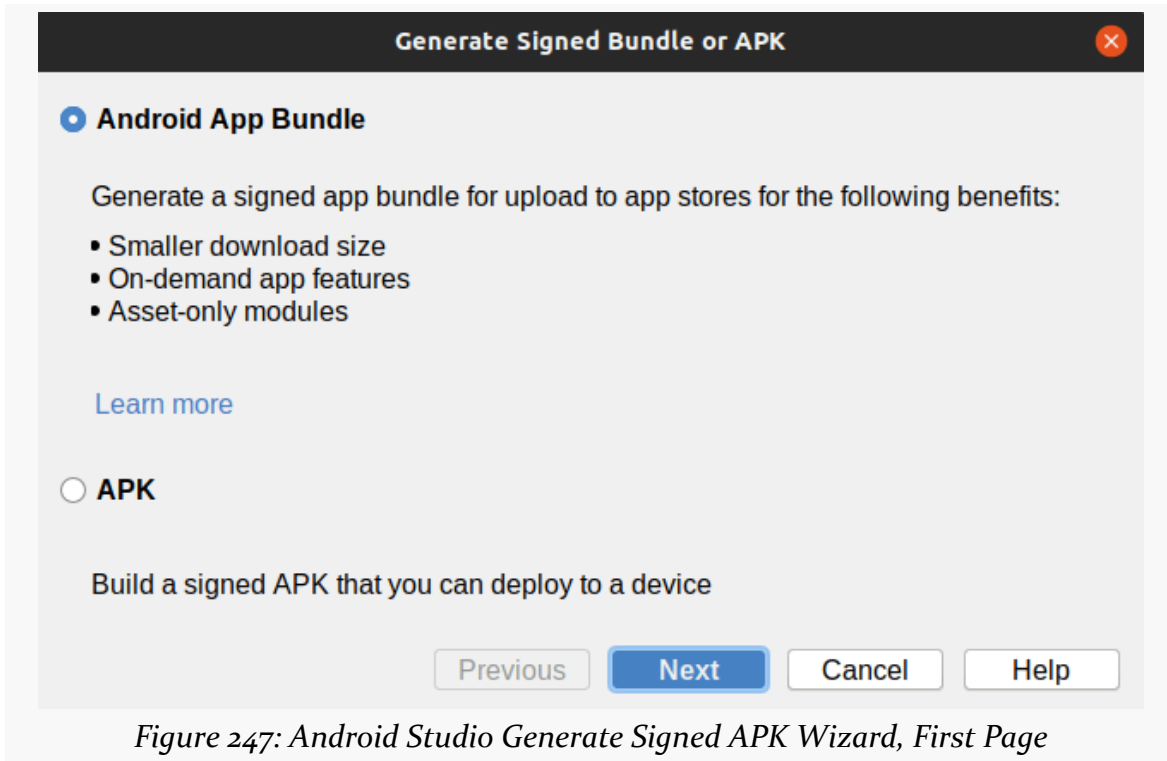


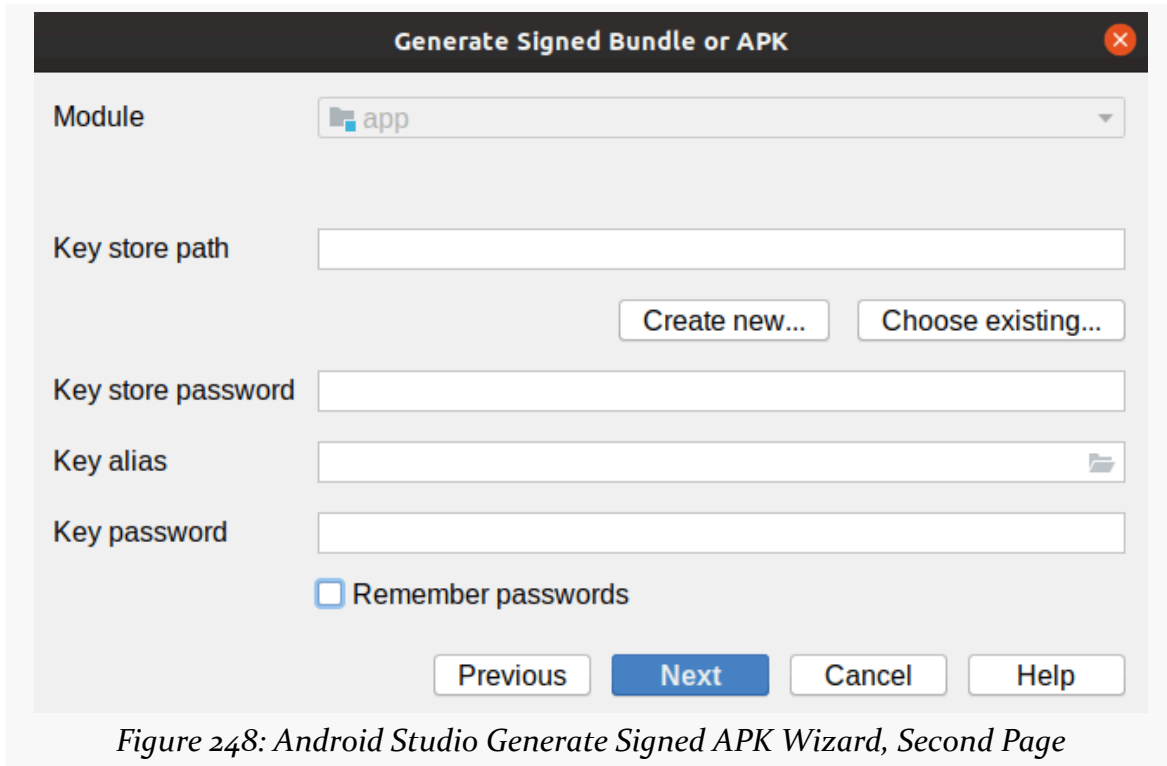
Figure 247: Android Studio Generate Signed APK Wizard, First Page

You have two options: an “Android app bundle” or an APK. While Google would like you to go with the “app bundle” route:

- You will be limited to distributing your app through the Play Store and nowhere else
- Google can modify your app as they see fit

SIGNING YOUR APP

This chapter will focus on the APK option. Choosing it and clicking “Next” will advance you in the wizard to where you can choose what to sign:



The drop-down at the top will let you choose a module from which to build your app. In most projects, there will be only one option, such as an app module.

The rest of the dialog is focused on getting your signing key, from a keystore file.

SIGNING YOUR APP

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:

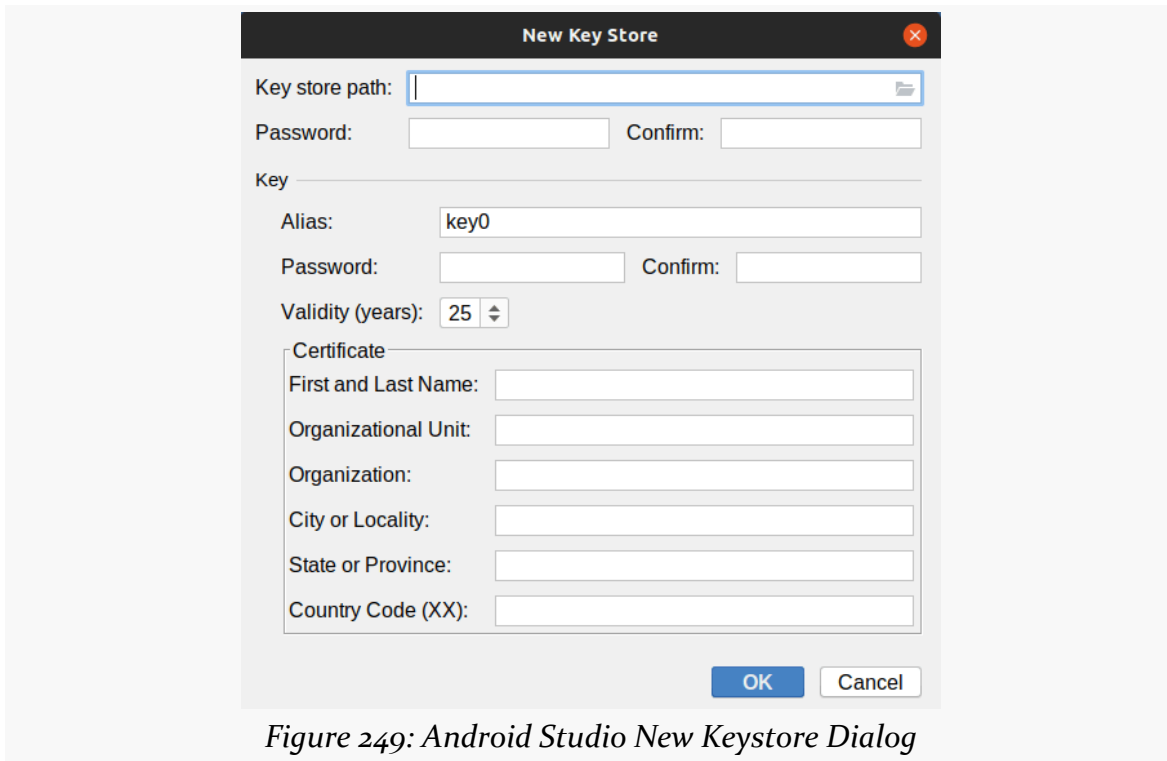


Figure 249: Android Studio New Keystore Dialog

You will need to provide a path to the keystore, manually or via the folder 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:

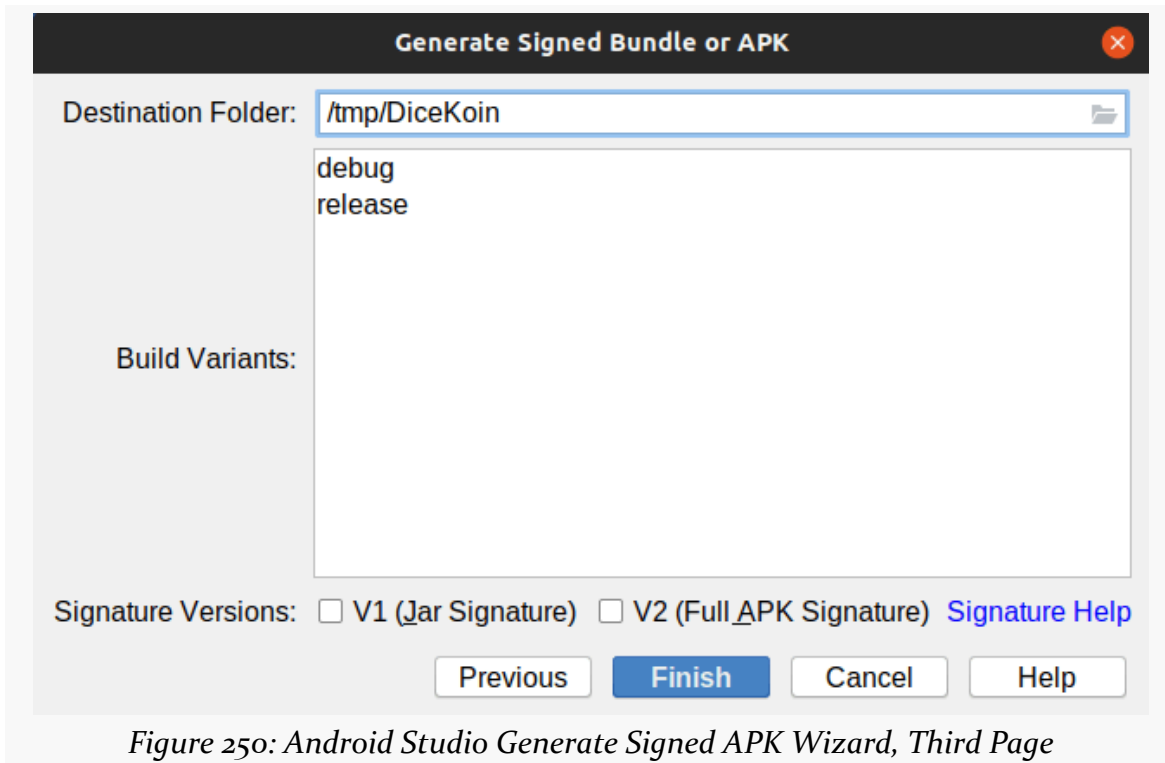
- “Alias” to indicate what human-readable name we want to give the key
- “Password” and “Confirm”, to specify a password for this specific key in the keystore (independent of the keystore’s own password)
- “Validity”, to indicate how long this key should be valid, where 25 years or more is recommended
- Details about you and your organization, asking for the standard information used in generating SSL-style keys

Clicking “OK” will generate the keystore file and save it where you specified. **Be sure to back up this keystore file** and safely record the passwords that you used.

SIGNING YOUR APP

If you already have a keystore file, 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 file. 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:



You can indicate where the APK file should be written and what build type to use (e.g., release).

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 popup will appear indicating that the work is completed. In the directory that you specified, Android Studio will create a subdirectory based on your build type (e.g., `release/`), and in there will place your signed APK file.

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.

Shrinking Your App

You might think that the app that you create is the smallest that it could be, by default. After all, why would a build process put extra stuff in your app that you are not using?

Unfortunately, that's exactly what happens.

As a result, the APK file that you get from the build process, or from [signing your app](#), may be larger than is necessary. In some cases, this is not a big problem. In other cases, though, it may be more of an issue.

So, in this chapter, we will explore what you can do to reduce the size of your APK.

Why We Care

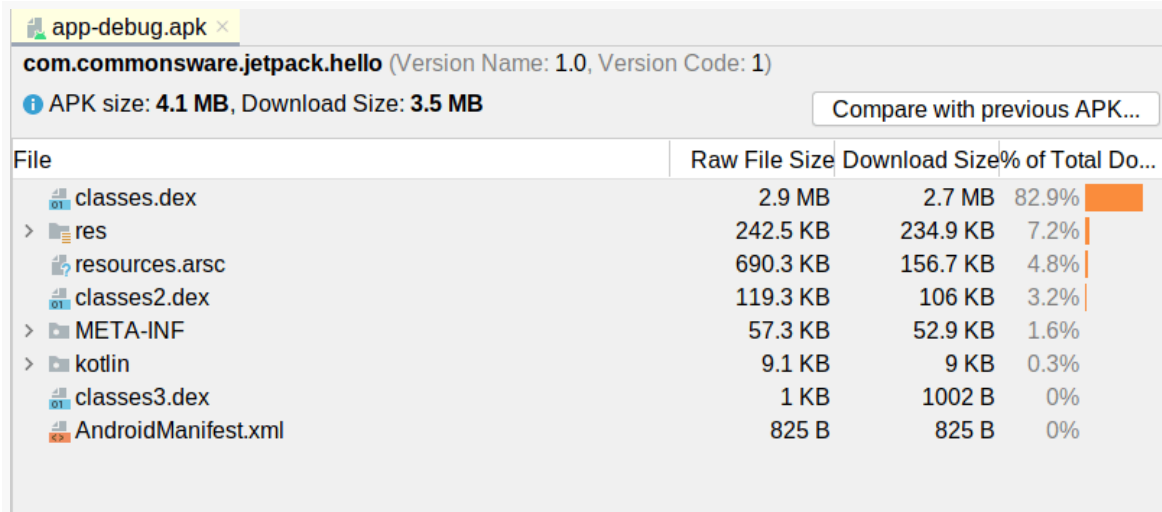
Your users may care about how much it costs to download your app, if they use a metered data connection and pay by the MB.

Your users may care about how much on-disk space your app consumes, if their device is short on available space. Note, though, that the amount of disk space that your app consumes is only partly from the APK — it also comes from any content that you store locally from within your app, such as in [Room databases](#).

You may care more directly, in that your chosen app distribution channels may impose limits on the APK size. Google's Play Store, for example, has a 100MB limit on the size of an APK.

Identify What to Attack

Back in [an earlier chapter](#), we looked at the APK Analyzer and the sorts of output it can provide:



The screenshot shows the APK Analyzer interface for a file named 'app-debug.apk'. The package is 'com.commonware.jetpack.hello' (Version Name: 1.0, Version Code: 1). The APK size is 4.1 MB and the download size is 3.5 MB. A button 'Compare with previous APK...' is visible. Below is a table listing the files and their sizes.

File	Raw File Size	Download Size	% of Total Do...
classes.dex	2.9 MB	2.7 MB	82.9%
res	242.5 KB	234.9 KB	7.2%
resources.arsc	690.3 KB	156.7 KB	4.8%
classes2.dex	119.3 KB	106 KB	3.2%
META-INF	57.3 KB	52.9 KB	1.6%
kotlin	9.1 KB	9 KB	0.3%
classes3.dex	1 KB	1002 B	0%
AndroidManifest.xml	825 B	825 B	0%

Figure 251: APK Analyzer Output

That particular app — the HelloWorld project from the earliest chapters — has about 250KB of resources, with the rest of the APK mostly made up of compiled code. For a “Hello, World” app, that is rather large, since the app does not really do anything.

In this case, most of the app size is coming from compiled code. In such a scenario, you will want to look at [removing unnecessary dependencies](#) and [otherwise shrinking your code](#). For cases where resources dominate the app space, [removing unnecessary dependencies](#) is still useful, but you will also want to see about [removing any unnecessary resources](#) and [shrinking your app's images](#).

Shrinking Your Dependencies

Your module's build.gradle file will have a dependencies closure listing all its dependencies. Lines with implementation, kapt, and api will be the dominant ones, and those reflect code and resources that are packaged into your APK. By contrast, things like testImplementation and androidTestImplementation only affect [test code](#) and will not be part of your APK.

Your app may have few or lots of dependencies:

```
    exclude 'META-INF/AL2.0'
    exclude 'META-INF/LGPL2.1'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation 'com.google.android.material:material:1.4.0'
    implementation "io.insert-koin:koin-android:3.1.2"
    implementation "androidx.room:room-runtime:2.3.0"
    implementation "androidx.room:room-ktx:2.3.0"
    kapt "androidx.room:room-compiler:2.3.0"
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
    testImplementation 'junit:junit:4.13.2'
    testImplementation "org.mockito:mockito-inline:3.12.1"
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"
    testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
    androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
}
```

(from [ToDoTests/build.gradle](#))

This module has nearly 20 dependencies that will go into the APK... and this is a fairly trivial app.

Plus, each dependency can pull in others, via what is called “transitive dependencies”. For example, the `androidx.room:room-ktx` dependency in that listing:

- Pulls in `androidx.room:room-runtime`, which pulls in...
- `androidx.sqlite:sqlite-framework` and `androidx.sqlite:sqlite`, which pull in...
- `androidx.annotation:annotation`
- And so on

If you fold open the “External Libraries” section of your project in the Android Studio explorer tree, you will see all the libraries that this project pulls in. This list will include test dependencies, so not everything in the list will go into your APK. But it also shows you the results of resolving all the transitive dependencies.

So... do you need all of that? Perhaps not.

Unfortunately, there is no good way to determine what you could remove. Perhaps by reviewing the dependencies list, you will recognize some that were from past experiments and are no longer needed.

The only real way to know if you can skip the dependency is to comment it out (e.g., using `//`), then see if your project builds and your tests run. If they do, presumably you can now remove the commented-out dependency entirely. If, on the other hand, there was some build failure, that means you are still referencing things from that dependency.

If very little of your app refers to things from that dependency, though, perhaps you could find another way of implementing that functionality that would let you remove the dependency. For example, `HelloWorld` uses `ConstraintLayout`, from the `androidx.constraintlayout` set of dependencies. However, `ConstraintLayout` is not needed for that simple UI — you could accomplish the same look with something else, such as a `FrameLayout`. Rewriting that layout resource to avoid `ConstraintLayout` would allow you to remove the `androidx.constraintlayout` dependencies, and that would reduce the size of the `HelloWorld` APK.

Shrinking Your Code

“Your code” refers to both the stuff that you write yourself and the stuff that comes from the dependencies that you need (including transitive dependencies).

Probably you use all the code that you wrote yourself. Otherwise, why did you write it?

However, it is very likely that you are not using all of the features from the dependencies that you are using. Any unused code is simply taking up space in your APK, and so it would be nice to get rid of it.

Unfortunately, that is a bit tricky.

A Tale of Two Tools

Finding unused code in a Java/Kotlin codebase requires examining all of the code, what it refers to, and from there determining what *isn't* referred to. Then, we need to actually remove the unused code from what goes into the APK, while not actually

affecting the dependencies themselves.

The original tool for this process was [ProGuard](#). ProGuard has been around for nearly two decades, and the Android build tools integrated it fairly early on to help reduce APK size.

More recently, Google has switched to their own tool, R8. R8 is actually the compiler, and it handles identifying unused code as part of the compilation process.

Enable Code Minification

However, by default, eliminating unused code (“minification”) is disabled. Partly, that is for build speed, as finding unused code is a time-consuming process. However, part of the reason why it is disabled is because sometimes the tools will *think* something is unused when in reality it is not.

To opt into code minification, we use `minifyEnabled true` in our module’s `build.gradle` file, typically only for the release build type:

```
buildTypes {  
    release {  
        minifyEnabled true  
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
    }  
  
    // other stuff here  
}
```

Here, we have a `buildTypes` closure, where we configure the release build. In there, we have `minifyEnabled true`, to override the default and ask that R8 try to remove unused code.

You are welcome to enable minification for debug builds as well, using a similar closure. Minification takes time, and so for larger projects it may become impractical to use minification for the builds that you want to do several times per day.

Test and Adjust

Once you enable minification, you will need to test your app, for whatever build types you elected to use `minifyEnabled true`. This means both your [automated tests](#) and manual tests.

SHRINKING YOUR APP

If you get `ClassNotFoundException`, `MethodNotFoundException`, or `NoSuchFieldError` crashes with your minified build, whereas you do not in a regular build, that indicates that minification went too far and removed things that you really are using.

The `proguard-rules.pro` file that (probably) is in your module's directory is a place to put rules that will be used by R8 (or ProGuard) to configure the minification process. In particular, `-keep` rules will tell R8 to keep classes, methods, or fields that it might otherwise try to remove:

```
-keep class net.sqlcipher.* { *; }  
-keep class net.sqlcipher.database.* { *; }
```

These lines, for example, tell R8 to keep classes, methods, and fields in the `net.sqlcipher` and `net.sqlcipher.database` packages. The `*` in the fully-qualified class name will match all classes in the package, and the `{ *; }` will match all methods and fields. Or, you can provide names of specific classes, etc. that you want to keep.

You will need to identify and create `-keep` rules like these to get R8 to not remove the things that, when removed, cause the aforementioned crashes.

Remove ABIs

Some of your dependencies might have “native code” (C/C++ libraries) in addition to Java/Kotlin code. These will show up in a `lib/` directory inside the APK Analyzer output:

File	Raw File Size	Download Size
▼ lib	868.5 KB	808.8 KB
▼ x86_64	229.2 KB	211 KB
libjdns_sd_embedded.so	208 KB	191.5 KB
libjdns_sd.so	21.2 KB	19.5 KB
▼ x86	223.5 KB	205.2 KB
libjdns_sd_embedded.so	203.9 KB	187.1 KB
libjdns_sd.so	19.6 KB	18.1 KB
▼ arm64-v8a	216.6 KB	202.4 KB
libjdns_sd_embedded.so	196.3 KB	183.7 KB
libjdns_sd.so	20.2 KB	18.6 KB
▼ armeabi-v7a	199.2 KB	190.3 KB
libjdns_sd_embedded.so	178.3 KB	170.3 KB
libjdns_sd.so	20.9 KB	20 KB

Figure 252: Native Code in APK Analyzer

This native code could be large — in this case, it is nearly 1MB. Frequently, the native code ships for multiple CPU architectures (or ABIs). In this case, we see four:

- x86_64
- x86
- arm64-v8a
- armeabi-v7a

The vast majority of Android devices use one of those latter two ABIs, for ARM CPUs (32-bit and 64-bit). Your emulator probably uses one of the x86 ABIs, but for a production release, you may be able to ignore the emulator.

You can eliminate native code for CPU architectures that you do not need by adding an `ndk` closure to your module’s `build.gradle` file, with an `abiFilters` declaration:

```
android {
    defaultConfig {
        // other stuff goes here

        ndk {
```

```
        abiFilters 'arm64-v8a', 'armeabi-v7a'  
    }  
}  
}
```

`abiFilters` is where you can list the ABIs that you want to keep. Others will be removed as part of the packaging process.

Removing Unused Resources

Some resources, like strings, are each fairly tiny. Some resources, like layouts, may be comparable in size to a source file. And some resources, like bitmaps, can be fairly large.

Ideally, building the app would get rid of any resources that you are not using. As with shrinking code, this should include both resources that you added to your project and resources that are being added by libraries.

And, as with code shrinking, removing unused resources is not automatic, but it is relatively easy to do.

Remove Manually

For your own project resources, you can remove them manually. Choose “Refactor” > “Remove Unused Resources” from the Android Studio main menu. This will display a dialog of options:

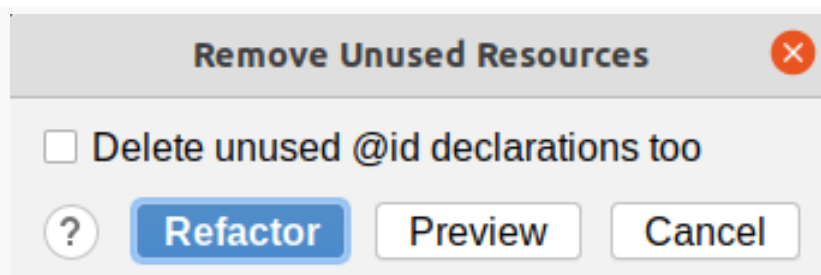


Figure 253: Android Studio Remove Unused Resources Dialog

SHRINKING YOUR APP

The safest thing is to leave the checkbox unchecked, then click the “Preview” button. This will show you what changes will be made, before they are made:

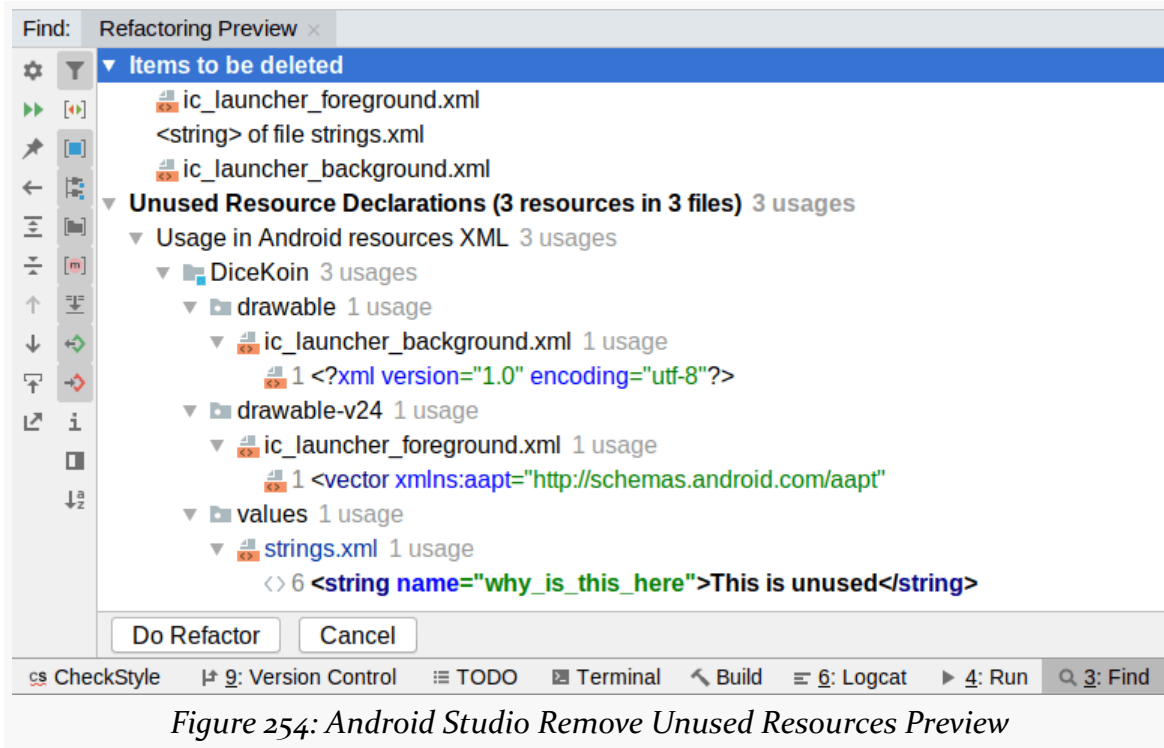


Figure 254: Android Studio Remove Unused Resources Preview

Here, it shows three resources that are unused: two drawables and a string.

Clicking “Do Refactor” will then remove those resources.

Remove Automatically

If you opted into code shrinking via `minifyEnabled true`, you can also opt into automatic resource shrinking via `shrinkResources true`:

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
    }
}
```

This works automatically, affecting your APK but not your project source code. It also removes library-supplied resources, whereas “Remove Unused Resources” does not. On the other hand, it needs to keep removing those resources on every build — “Remove Unused Resources” can improve your build speed a bit.

Whitelist Resource Sets

Libraries may supply resources that your app is unlikely to ever use.

For example, Jetpack libraries that have string resources often will supply string resource translations for every language, as the Jetpack developers do not know what languages your app will support. By default, the build tools also do not know what languages your app will support, so the build tools will package all translations. *You* may know what languages your app will support, in terms of the string translations that you intend to provide. By teaching the build tools about that language list, the build tools can strip out translations in other languages that your app is not going to use.

To do this, in the `defaultConfig` closure of your module’s `build.gradle` file, you can list language prefixes in a `resConfigs` list:

```
android {  
    defaultConfig {  
        // other cool stuff goes here  
  
        resConfigs "en", "es", "zh"  
    }  
}
```

Here, we indicate that we want the app to contain English, Spanish, and Chinese resources. Resources targeting other languages will be removed from the APK automatically by the build tools.

Optimizing Bitmaps

Typically, your largest resources are bitmaps: PNGs, JPEGs, etc. If you can remove some — such as through the techniques in the preceding sections — that is great! For those that remain, though, you still have some options for making them take less space than they do presently.

Switch to Vectors

You may be able to replace some of those bitmaps with vector drawables, the ones we used in this book for toolbar button icons.

For example, you may get bitmaps to put in your app from a graphic designer. If the graphic designer can supply you with an SVG image instead, you can use the Vector Asset wizard to try converting that SVG to a vector drawable. If that drawable looks good, you can remove the bitmaps.

A vector drawable on its own typically is smaller than an equivalent PNG file. And a vector drawable is density-independent, so you only need one such drawable to get good results on all screen densities. With bitmaps, you might have multiple images in multiple drawable directories (e.g., `res/drawable-hdpi/`, `res/drawable-xhdpi/`). Replacing several bitmaps with a single vector drawable should result in a substantial reduction in your APK size.

However, do not try swapping your launcher icon mipmap with a vector drawable, as not all launcher apps are set up to handle that.

Reduce Resolution

The biggest bitmaps of all tend to be photographs. Modern cameras offer very high resolution, and high-resolution photos take up a lot of disk space.

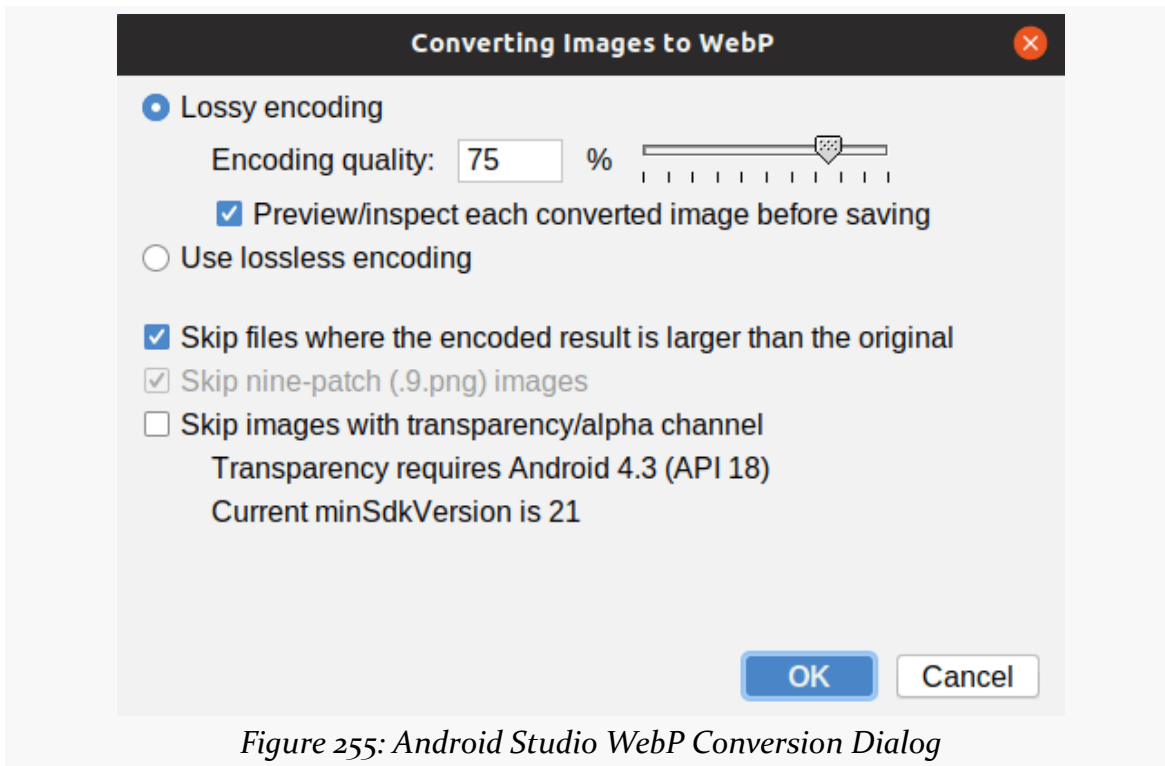
Moreover, you may not need all of that resolution. For example, it may be that you only ever show the image as a thumbnail. In that case, you do not need a 20-megapixel photo — you could safely reduce the resolution to be a few hundred pixels on a side, rather than a few thousand. This will substantially reduce the size of those images.

Convert to WebP

Several years ago, Google introduced the WebP image format. This format uses a flexible compression algorithm that supports both lossless compression (like PNG) and lossy compression (like JPEG). A WebP image may be somewhat smaller than its PNG or JPEG counterpart.

If you right-click over a PNG or JPEG image in your project, you can choose “Convert to WebP...” from the context menu. This will bring up a dialog to configure the

conversion:



If the image is similar to a photo, you should choose “Lossy encoding”. If the image is an icon or other types of “line art”, choose “Lossless encoding”. If you have “Skip files where the encoded result is larger than the original” checked, you are assured that you will only get a WebP image from the conversion if that image will reduce your APK size. The “Skip images with transparency/alpha channel” usually can be unchecked, as this only affects apps that support Android 4.2 and older devices.

SHRINKING YOUR APP

If you chose “Lossy Encoding”, there is a checkbox to ask for a preview of the changes. If you choose that, you will get a preview window showing the original image and the WebP converted image, so you can see how they compare:

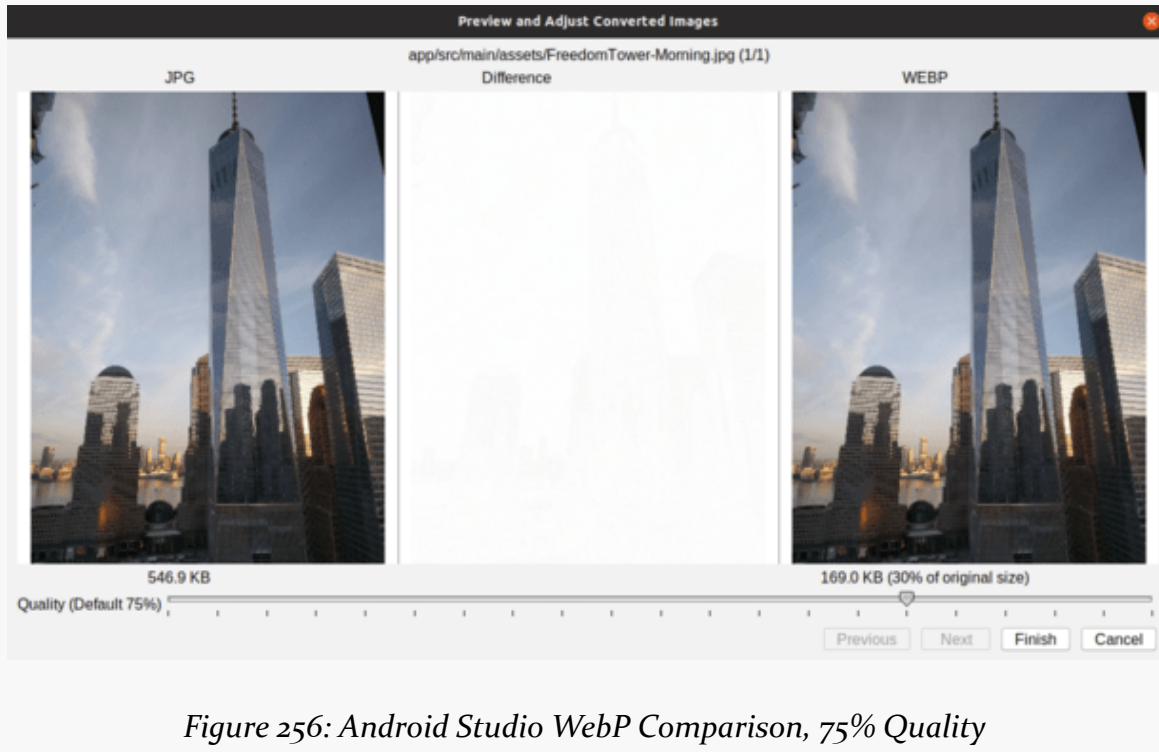


Figure 256: Android Studio WebP Comparison, 75% Quality

SHRINKING YOUR APP

Here, with a 75% WebP quality factor, we wind up with an image that is only 30% the size of the original, with acceptable quality. The “difference” area in the center shows a comparison of what pixels changed between the images, and by how much. A very low quality factor starts making more significant changes to the image, for example:

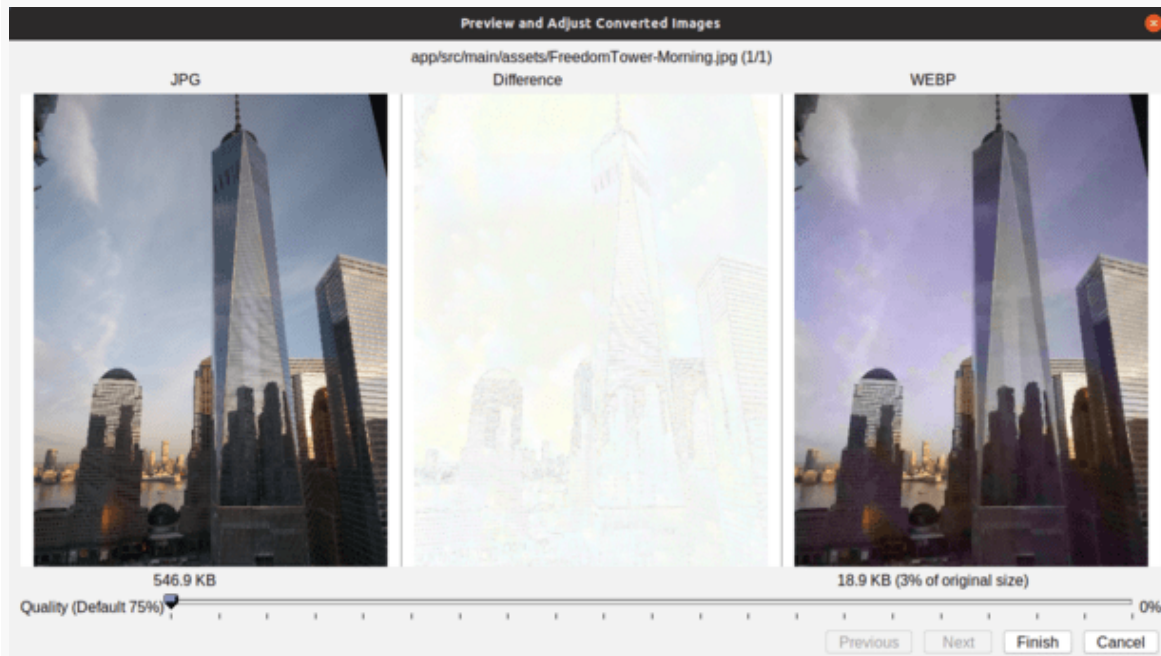


Figure 257: Android Studio WebP Comparison, 1% Quality

(if you look very closely at that middle area, you will see some pixels showing up)

Once you “Finish” the WebP conversion wizard, your image will be converted to WebP (only if actually reduce the size, if you checked that one checkbox).

Crunch Your PNGs and JPEGs

If you would prefer to keep your PNGs and JPEGs in their original form, rather than convert them to WebP, there are a variety of tools available to “crunch” them down to take up less space. pngquant, jpegoptim, optipng, and others are available for use from the command line. Web-based equivalents also exist. There are even Gradle plugins that can automate the process.

Remove Densities

In principle, you could use `resConfigs` to limit the densities that you support, just as you use it to limit the languages that you support. If you try this, you will want to test your app on a variety of screen densities and ensure that your images still look OK.

Hey, What About App Bundles?

Google's solution for all of this is the App Bundle.

If you upload an App Bundle, instead of an APK, to the Play Store, Google will deliver only the pieces that a given user needs:

- Instead of all image densities, Google will send down the densities that make sense for the user's device
- Instead of all the languages, Google will send down the languages that make sense for the user's device
- Instead of all the ABIs for different CPU architectures, Google will send down the ABI(s) that make sense for the user's device

In other words, they handle a lot of the optimization described in this chapter for you automatically.

The downside is that Google has to sign your app. Once you do that, you no longer control what Google ships. While Google's public focus is on what they remove (e.g., unused image densities), Google is perfectly capable of adding or replacing other things in your app... things that you might not want to be added or replaced.

If you trust Google completely and wish to use app bundles, you are welcome to do so. If you do not trust Google completely, ship APKs that you digitally sign and take your own steps to shrink them down to size.

Using the AVD Manager and the Emulator

Some developers focus on using actual Android-powered devices for testing their Android apps. However, nearly every developer winds up using the Android SDK emulator for at least some work. We [set up the emulator](#) and [set up an AVD](#) earlier in the book. Here, we will explore some more capabilities of the emulator, beyond the basics of being able to run your Android app.

Notable AVD Configuration Options

When defining an AVD, or editing an existing AVD definition, there are many configuration options at your disposal, beyond those that we saw in the earliest chapters of the book.

Hardware Graphics Acceleration

One 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.

USING THE AVD MANAGER AND THE EMULATOR

This setting is toggled within the AVD Manager, for new and existing AVDs, via the “Graphics” drop-down list in the “Emulated Performance” group:

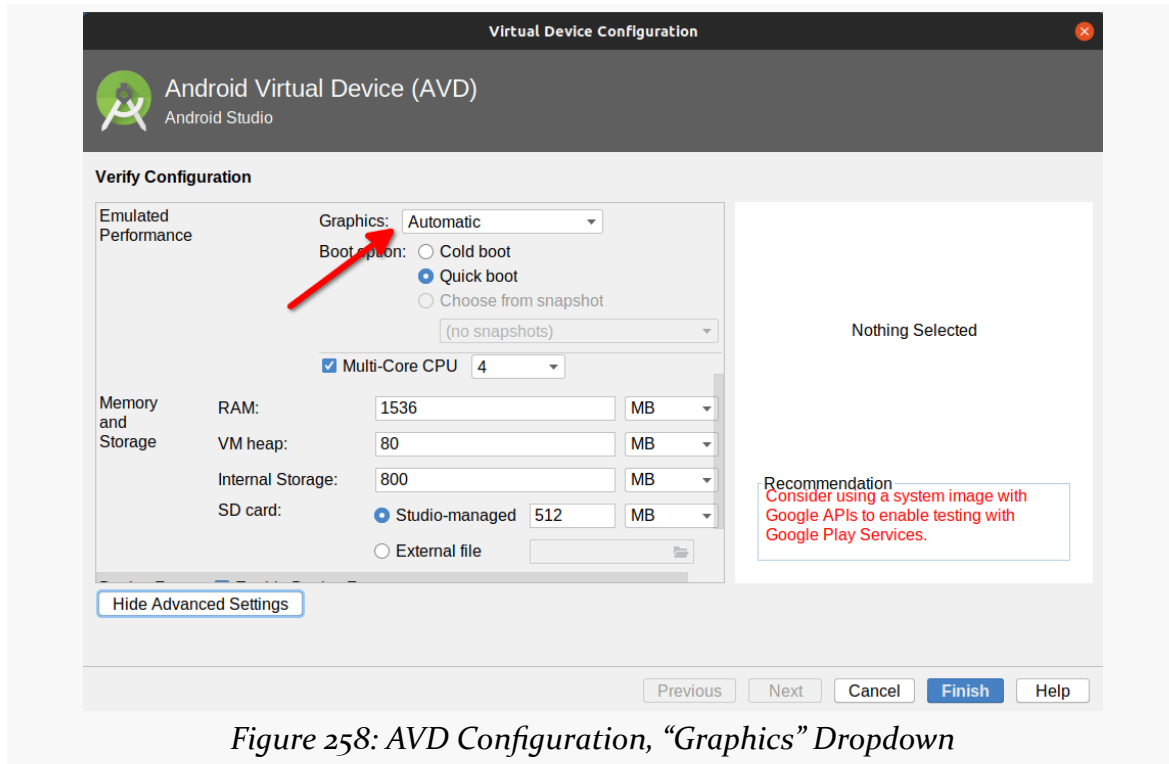


Figure 258: AVD Configuration, “Graphics” Dropdown

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

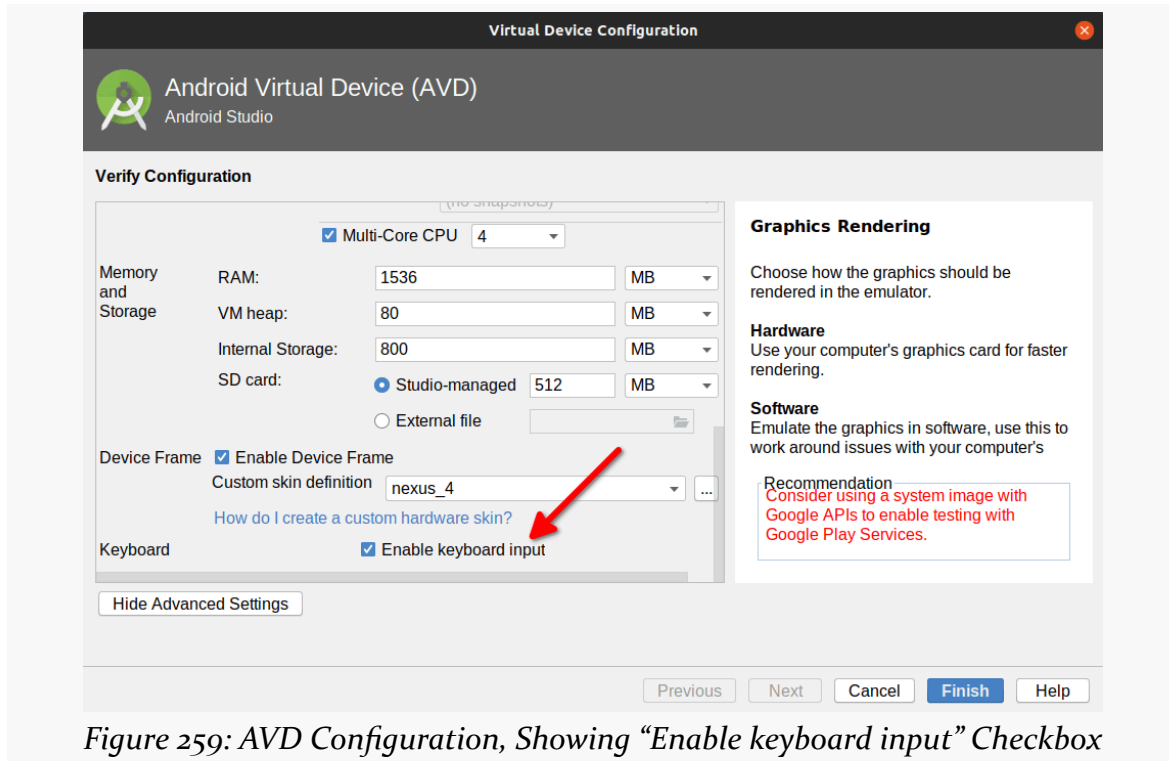
The Android emulator can emulate devices that have, or do not have, a physical keyboard. Most Android devices do not have a physical keyboard, and so the emulator is set up to behave the same. However, this means that typing on your development machine’s keyboard will not work in `EditText` widgets and 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 —

USING THE AVD MANAGER AND THE EMULATOR

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:



Startup Settings

You can also control whether the device starts up in portrait or landscape mode at the outside, by the toggle buttons labeled “Startup orientation”:

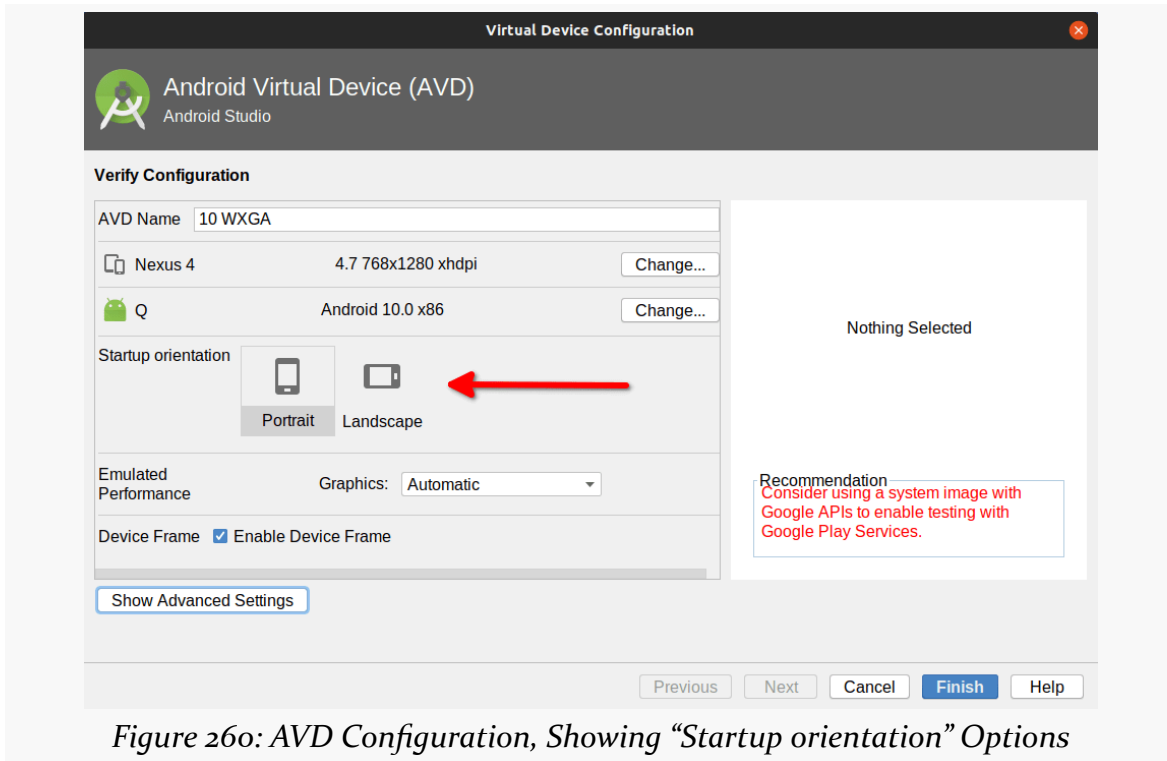
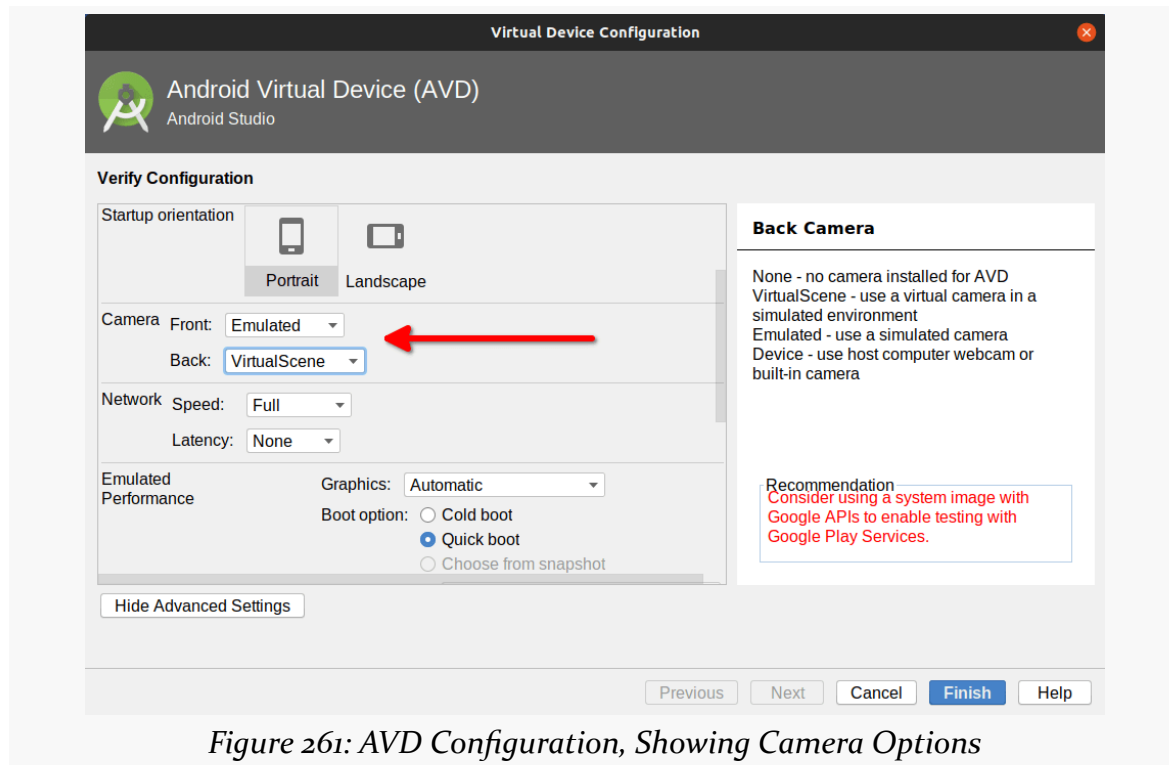


Figure 26o: AVD Configuration, Showing “Startup orientation” Options

Camera Options

In the “Advanced Settings” area, you can control whether or not the emulator emulates a device with a camera:



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., Webcam0), 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)

The back camera also will have a “VirtualScene” option, where the camera will appear to be looking at the interior of a home, in the form of a 3D rendering of that interior.

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:

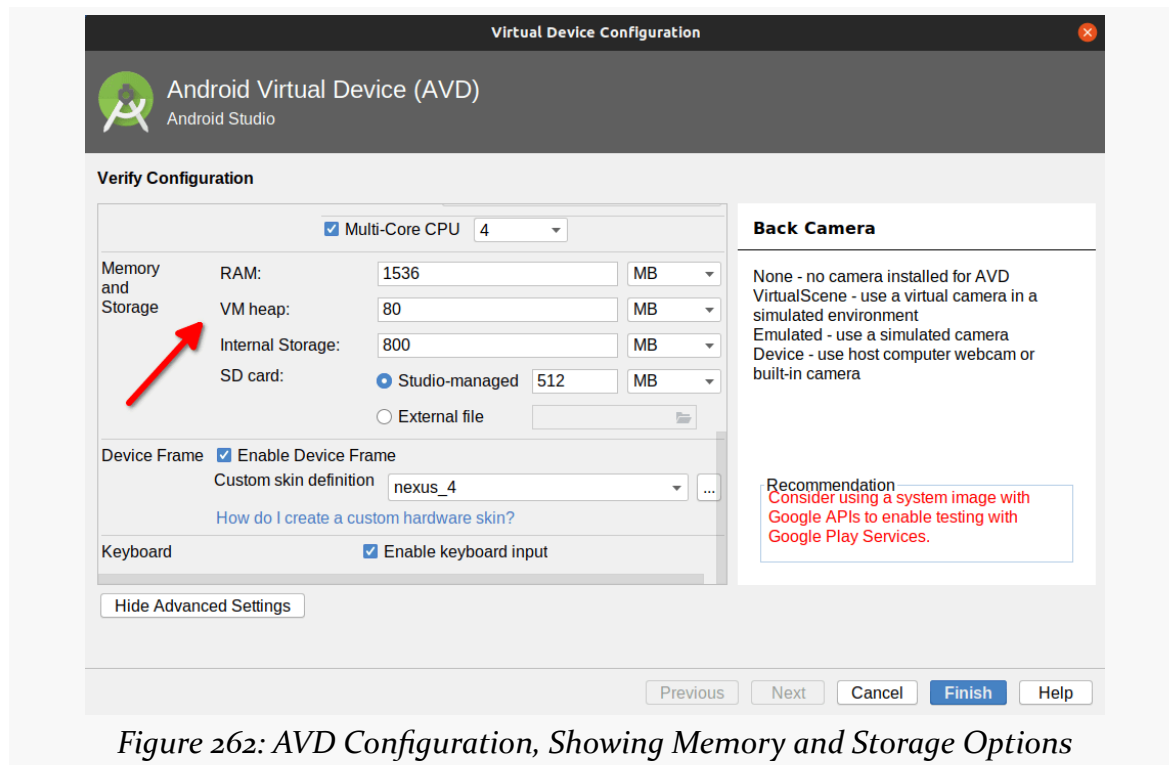


Figure 262: AVD Configuration, Showing Memory and Storage Options

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 `mktdcard` to create your own disk image that you attach to the emulator.

Usually, the defaults are fine.

The Emulator Sidebar

The free-standing emulator sports a “sidebar” that runs alongside the main emulator window:



Figure 263: 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.

USING THE AVD MANAGER AND THE EMULATOR

If you click the “More” button, you will open up the “Extended Controls” window:

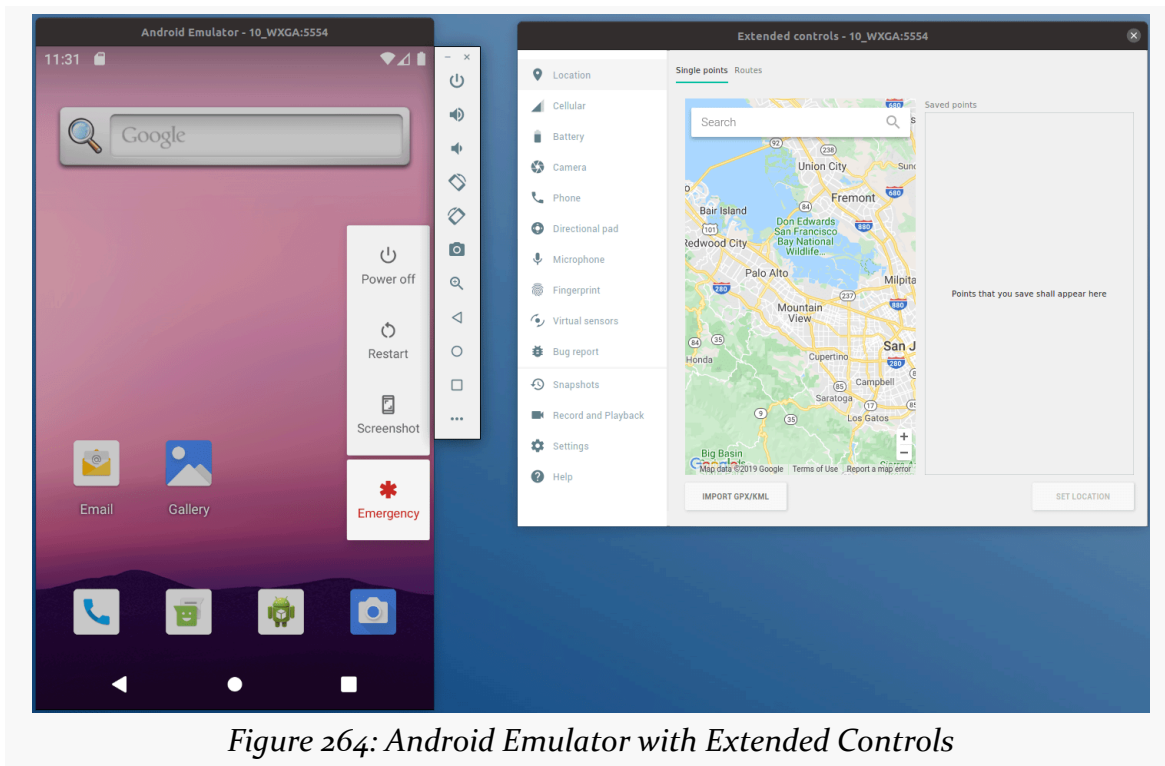


Figure 264: Android Emulator with Extended Controls

Some of the features are configured via this window.

Power and Navigation Controls

The top icon in the sidebar is a power button. A quick click on it will simulate putting your emulator to sleep; clicking it again will “wake it up”. A long-click will behave like the POWER button on an Android device, bringing up the power menu:



Figure 265: Android Emulator, Showing Power Menu

USING THE AVD MANAGER AND THE EMULATOR

Towards the bottom of the sidebar are BACK, HOME, and RECENTS buttons for navigation:

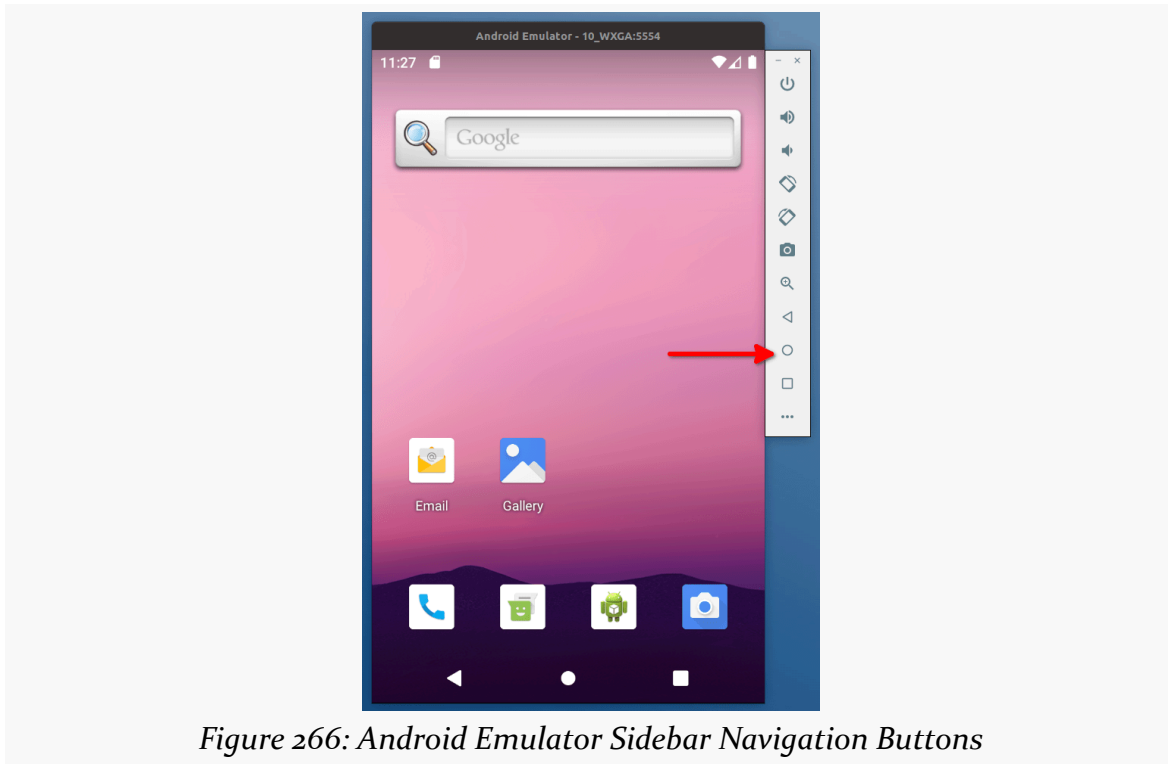


Figure 266: Android Emulator Sidebar Navigation Buttons

Screen Orientation

Two buttons on the sidebar allow you to rotate the device clockwise or counter-clockwise:

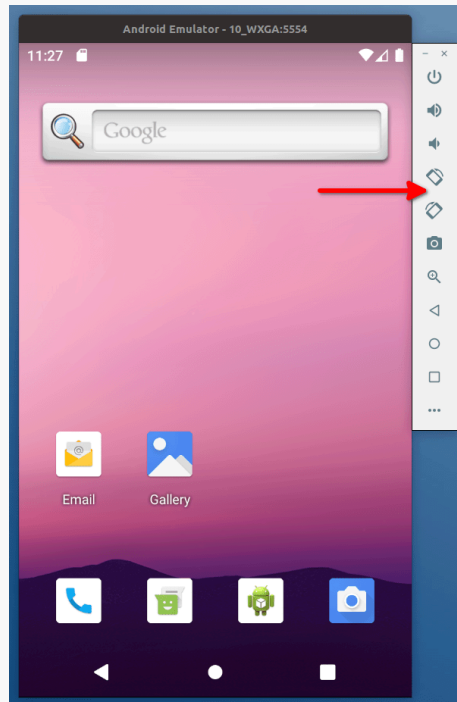


Figure 267: Android Emulator Sidebar Rotation Buttons

USING THE AVD MANAGER AND THE EMULATOR

This allows you to rotate between all four portrait and landscape orientations:

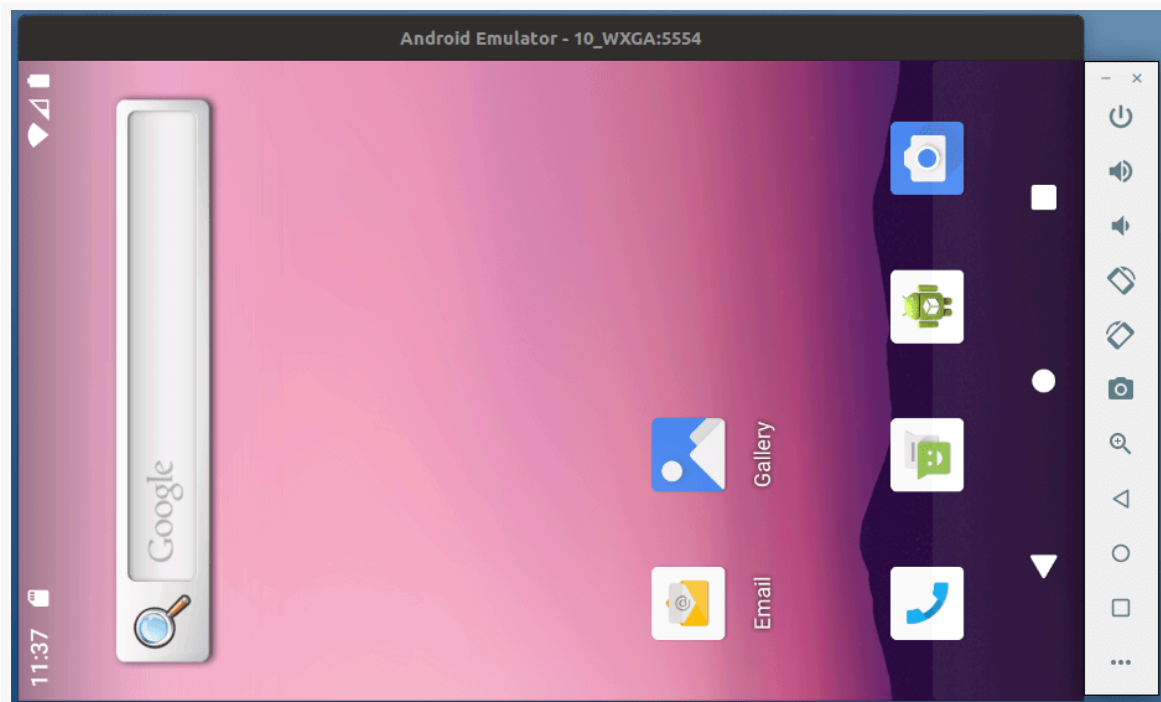


Figure 268: Android Emulator in Landscape

Screenshots

The camera button on the sidebar allows you to rapidly take screenshots of the emulator window:

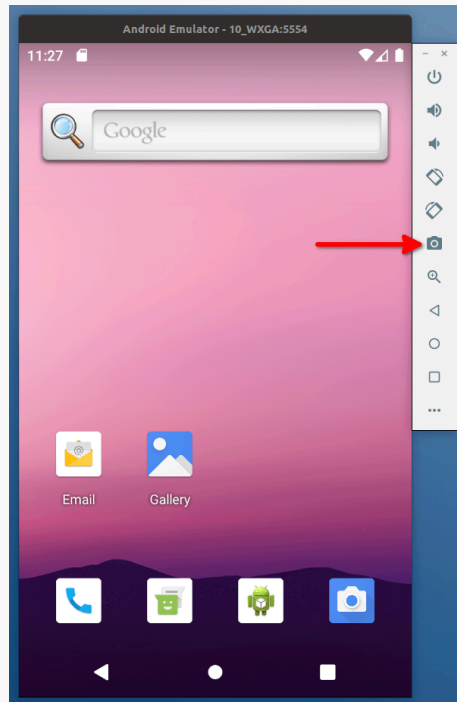


Figure 269: Android Emulator Sidebar Screenshot Button

USING THE AVD MANAGER AND THE EMULATOR

These will be stored in a directory controlled by the “Settings” category in the “Extended controls” window:

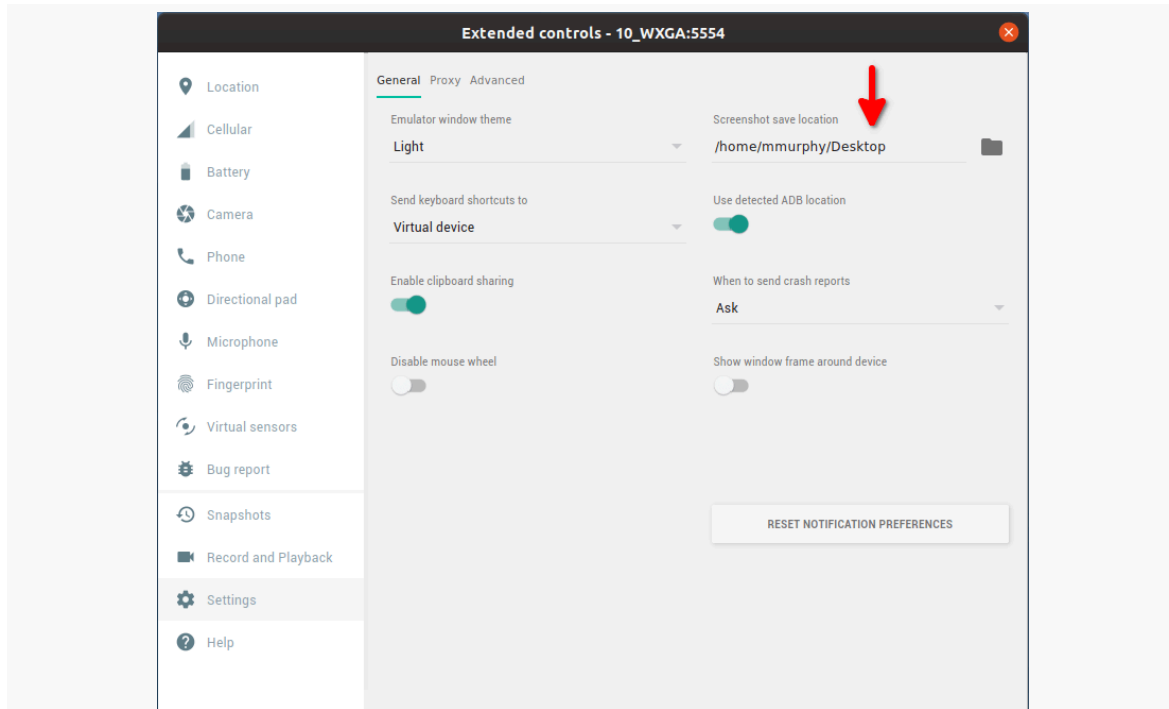


Figure 270: Extended Controls, Showing Screenshot Save Location in Settings

Faking the Real World

That “Extended controls” window also allows you to fake real world behavior in your emulator.

Location

The “Location” category lets you fake GPS fixes, if your app winds up using locations.

There are two tabs: “Single points” and “Routes”. As the names suggest, you use the “Single points” tab to set the emulator GPS to a specific point, and you use “Routes” to have the emulator play back a series of GPS fixes with time intervals between them.

For specifying points for GPS fixes, the emulator needs a latitude and longitude. Older versions of the emulator let you provide those values directly. Now, instead,

USING THE AVD MANAGER AND THE EMULATOR

you get a tiny Google Map and need to provide locations that way:

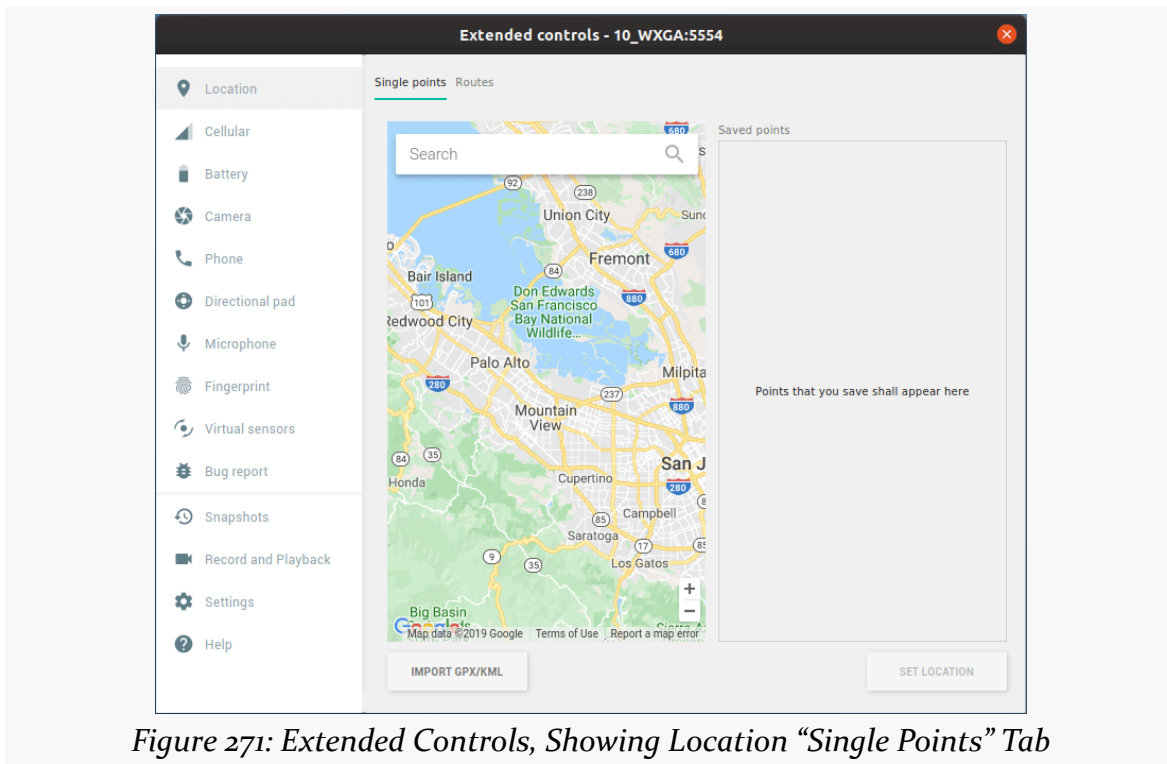


Figure 271: Extended Controls, Showing Location “Single Points” Tab

You can pan around the map by dragging it using your mouse, and you can zoom it using the +/- buttons. You can search for a location using the “Search” field, as you would on the Google Maps Web site or app.

USING THE AVD MANAGER AND THE EMULATOR

A simple click will place a marker pin at the clicked point, showing you the address:

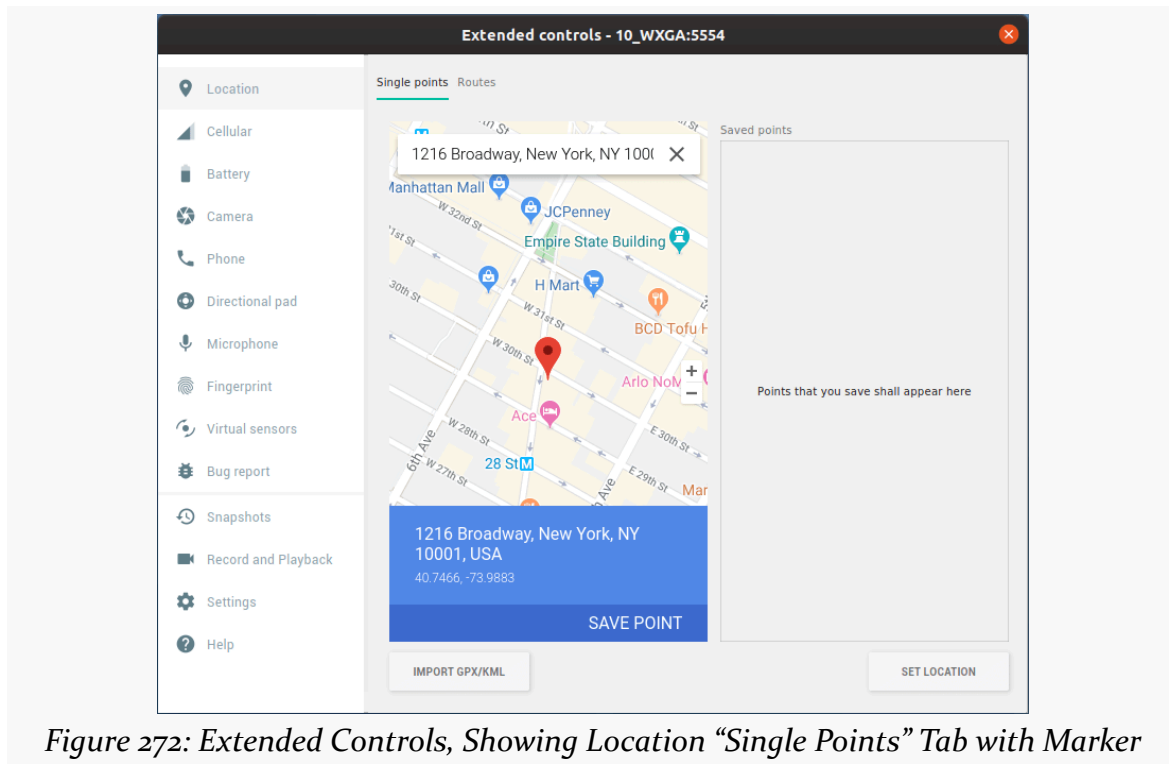


Figure 272: Extended Controls, Showing Location “Single Points” Tab with Marker

USING THE AVD MANAGER AND THE EMULATOR

The “Set Location” button in the corner of the window will set the simulated GPS fix to that location. The “Save Point” option in the map window itself will add this location to a list of saved points to the side of the map, under a name that you provide:

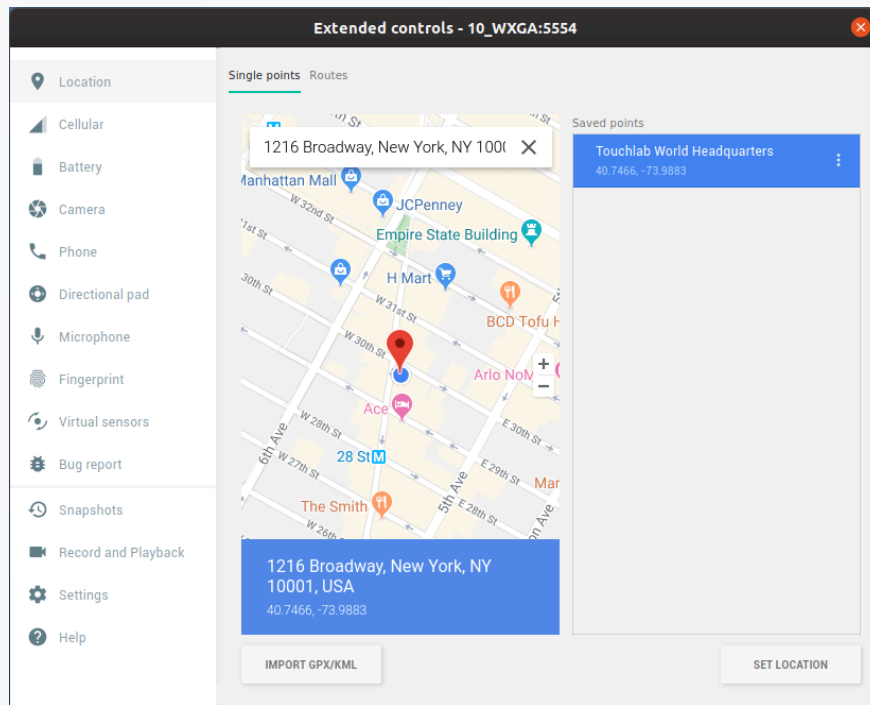


Figure 273: Extended Controls, Showing Location “Single Points” Tab with Saved Point

Later, you can click on items in the list to move the map to that location, then click the “Set Location” button to send that GPS fix.

USING THE AVD MANAGER AND THE EMULATOR

The “Routes” tab lets you build up a route from saved points. To start a route, you need to first save a point in the “Single points” tab, then choose “Start a Route” from the “...” drop-down menu. This will bring up a typical Google Maps navigation form, where you can specify start and end addresses:

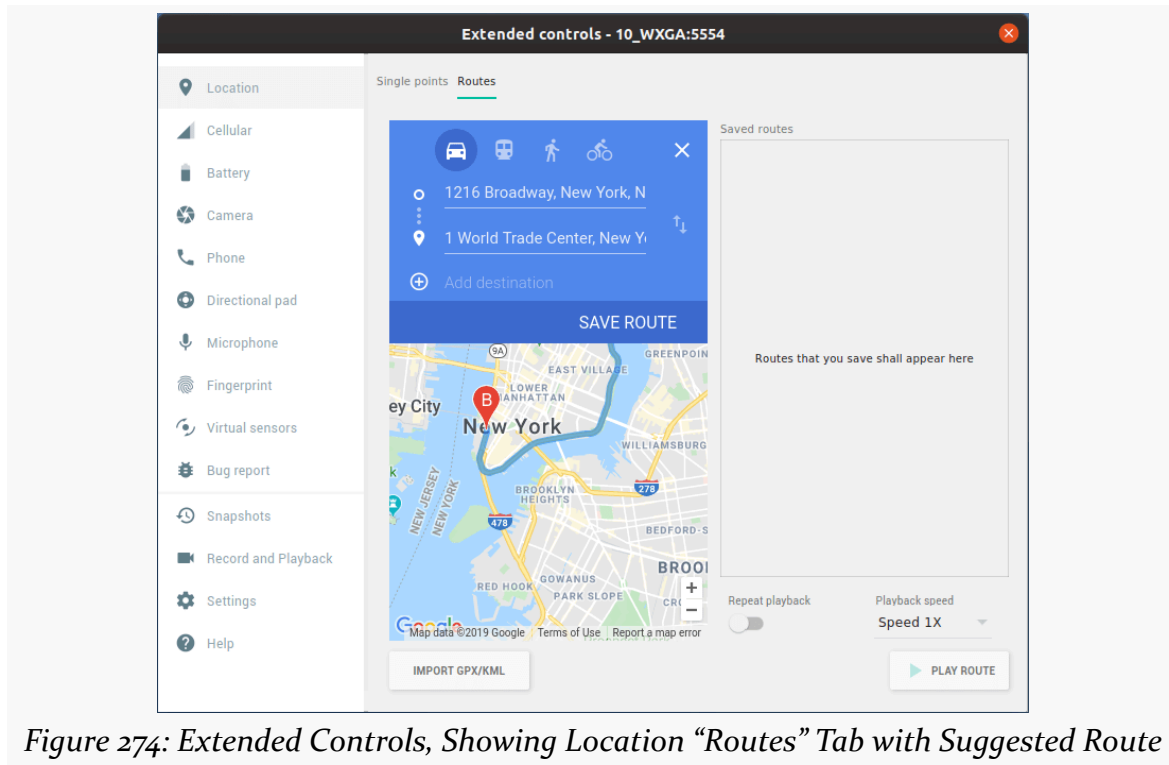


Figure 274: Extended Controls, Showing Location “Routes” Tab with Suggested Route

Clicking “Save Route” will save it to the list on the right:

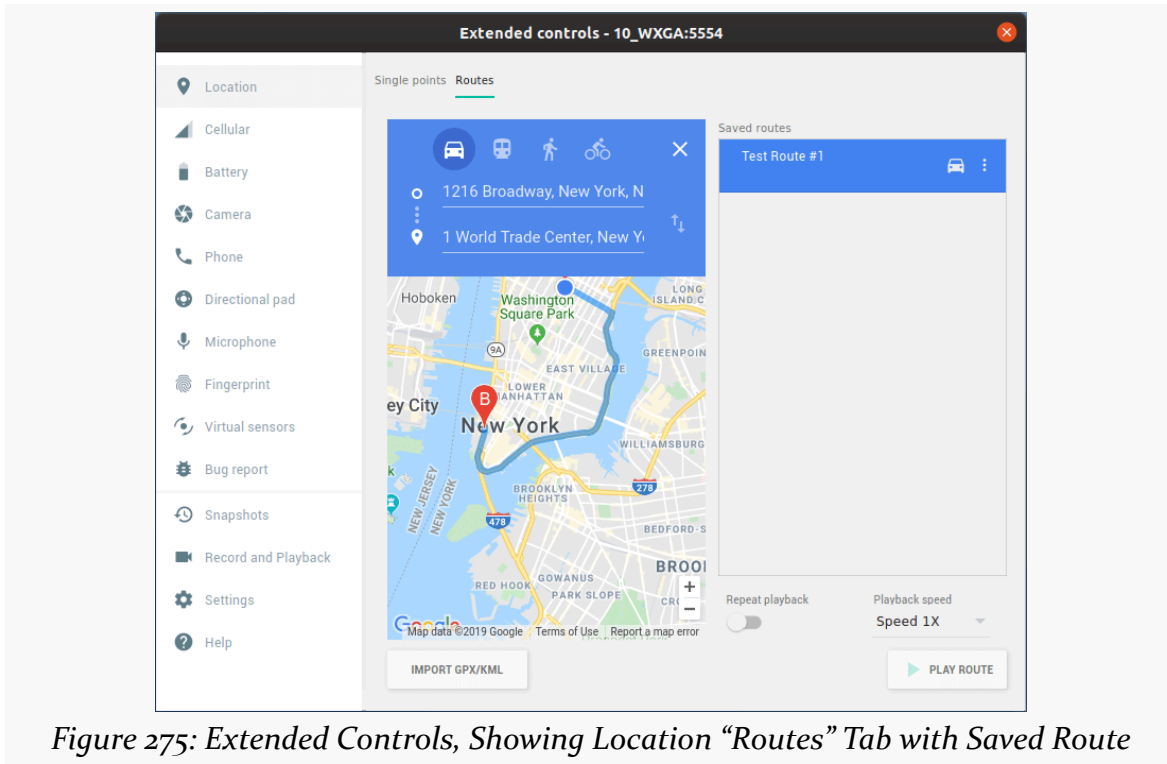


Figure 275: Extended Controls, Showing Location “Routes” Tab with Saved Route

The “Play Route” button will then simulate the device moving along the map-supplied route, using real-world speeds for that particular route. The “Playback speed” drop-down lets you speed up the playback, if you get bored waiting for NYC traffic.

If you have a route in GPX or KML format, you can import that using the “Import GPX/KML” button. This allows you to use more sophisticated off-emulator tools to build up your route.

Network Status

The “Cellular” category controls how the emulator emulates its cellular network connection:

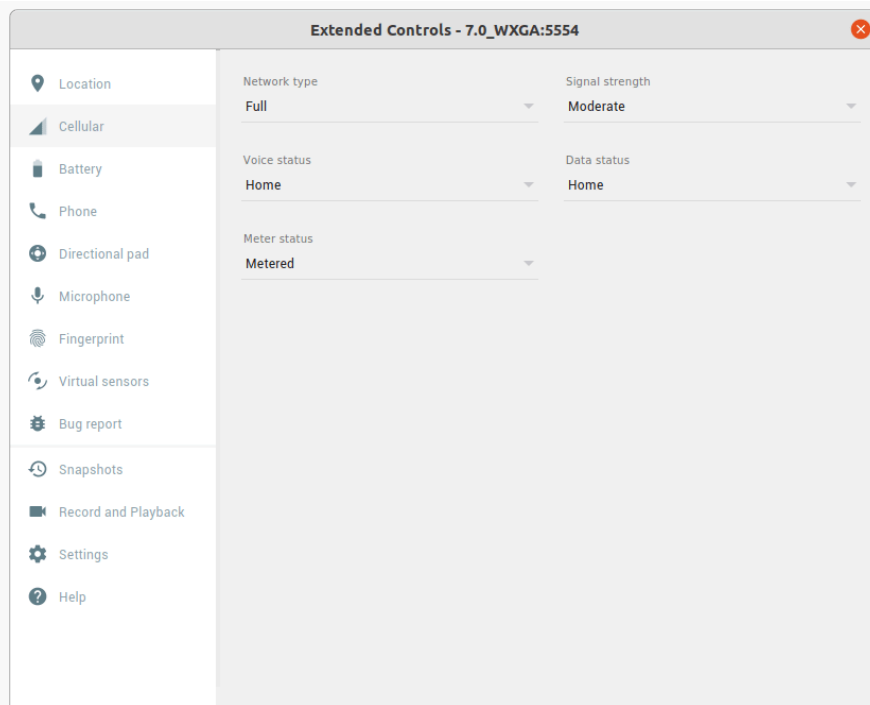


Figure 276: Extended Controls, Showing “Cellular” Options

Telephony

The “Phone” category allows you to simulate incoming phone calls and text messages:

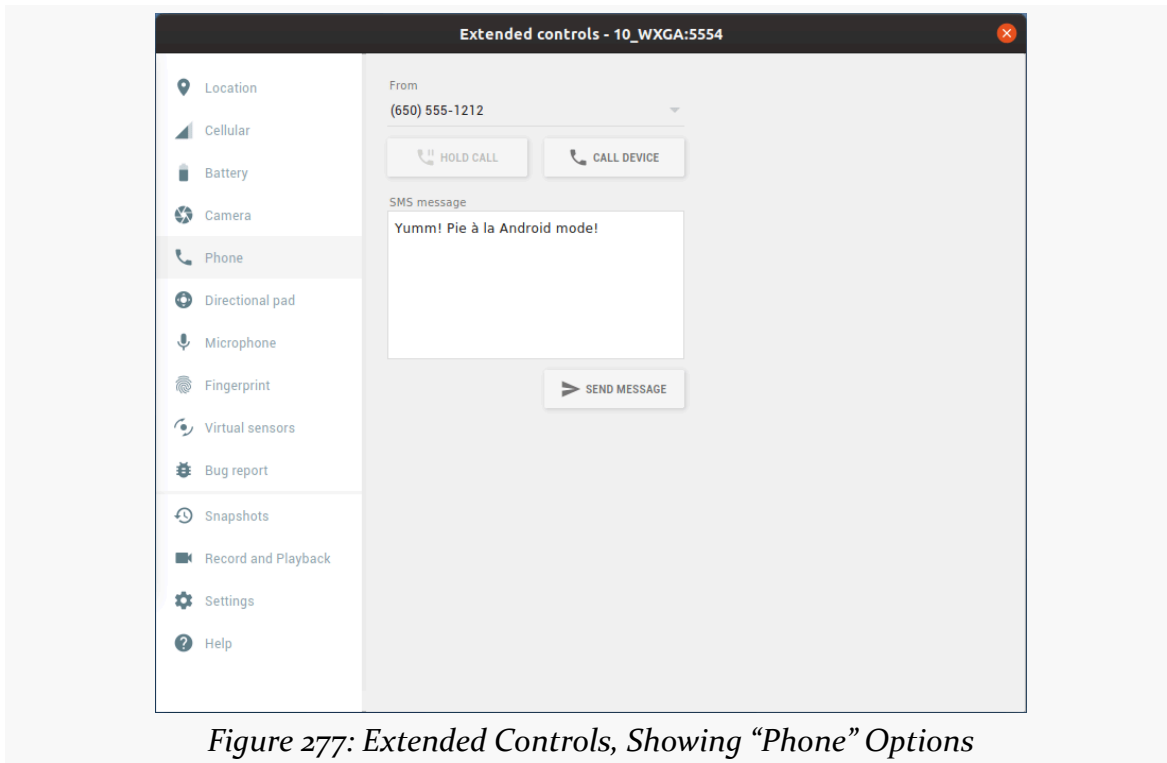


Figure 277: Extended Controls, Showing “Phone” Options

USING THE AVD MANAGER AND THE EMULATOR

These will trigger the corresponding apps on the emulator, based on registered Intent filters, such as responding to an incoming call:

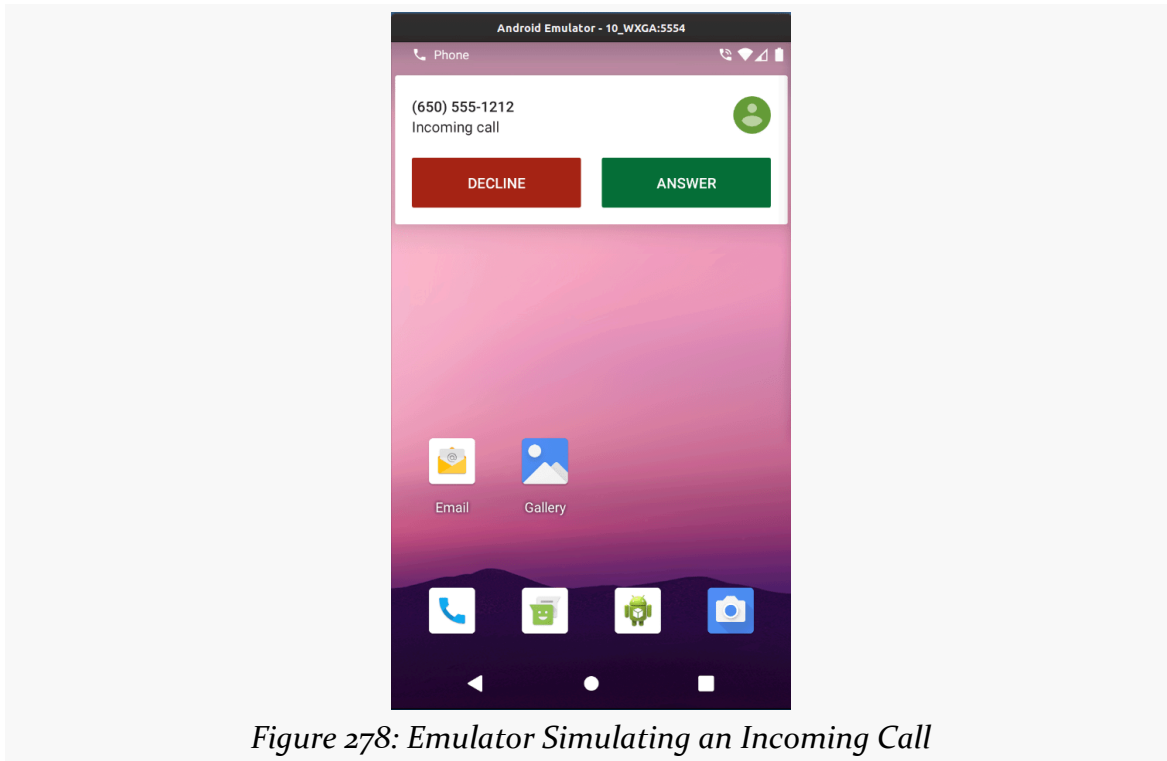


Figure 278: Emulator Simulating an Incoming Call

Miscellaneous Features

Additionally, you have:

- A “Battery” section allows you to simulate changes in the power status of the emulator
- A “Camera” section for controlling what the emulated camera shows, if it is not using a webcam
- A “Directional pad” section for simulating arrow keys and media buttons
- A “Microphone” section for controlling how the emulator simulates a headset, including whether your development machine’s microphone should be used as test input
- A “Fingerprint” section for simulating the user pressing their finger on a fingerprint sensor
- A “Virtual sensors” section for simulating rotating, shaking, twisting, or otherwise moving the device
- A “Bug report” section for submitting an issue to the Android issue tracker,

based on emulator behavior

- A “Snapshots” section for you to save different versions of the emulator AVD, representing different device states
- A “Record and playback” section for recording screencasts and playing back a series of touch inputs
- A “Settings” section for controlling other aspects of the emulator behavior, such as where to save screenshots
- A “Help” section with a list of keyboard shortcuts, links to documentation, etc.

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. 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.

In-IDE Emulator

Historically, the emulator has run in a separate window, as has been shown in this chapter and earlier in the book. Android Studio 4.1 added the option for having the emulator be embedded directly in the IDE window itself:

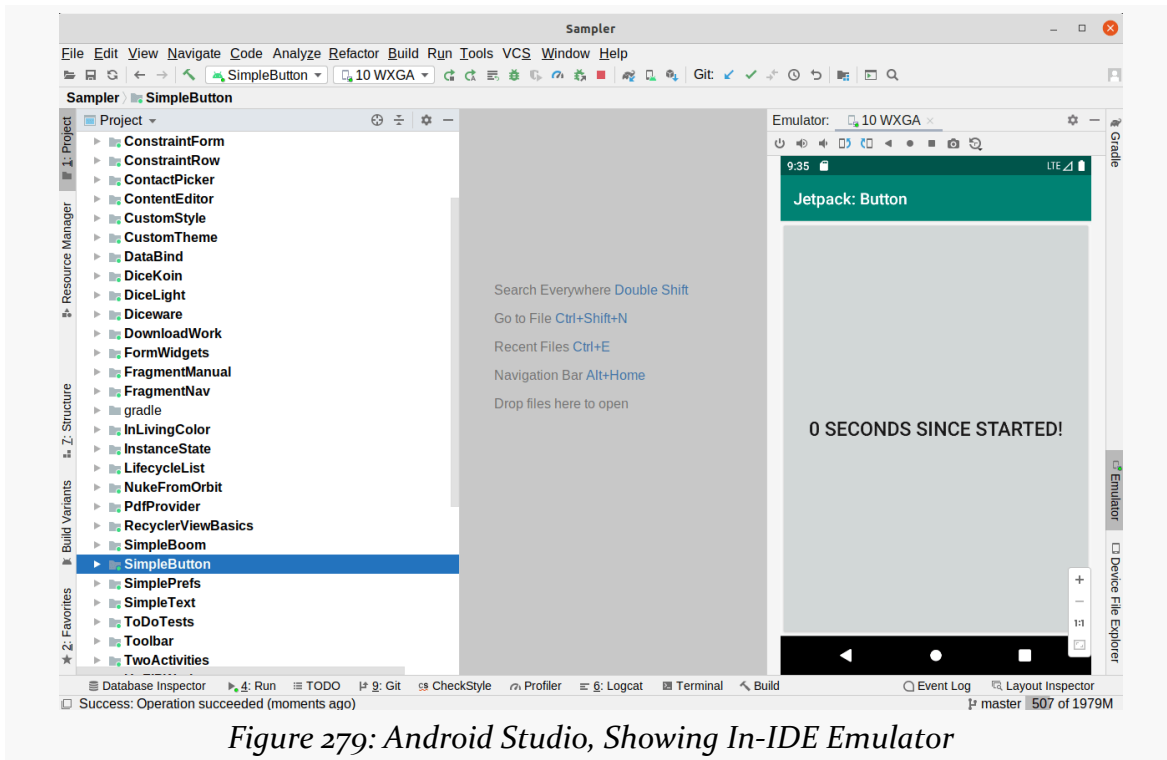


Figure 279: Android Studio, Showing In-IDE Emulator

USING THE AVD MANAGER AND THE EMULATOR

There is an “Emulator” tool, by default docked on the right. When you tap it, an Emulator tool window will open up... mostly to show you a message about why you are not seeing an emulator:

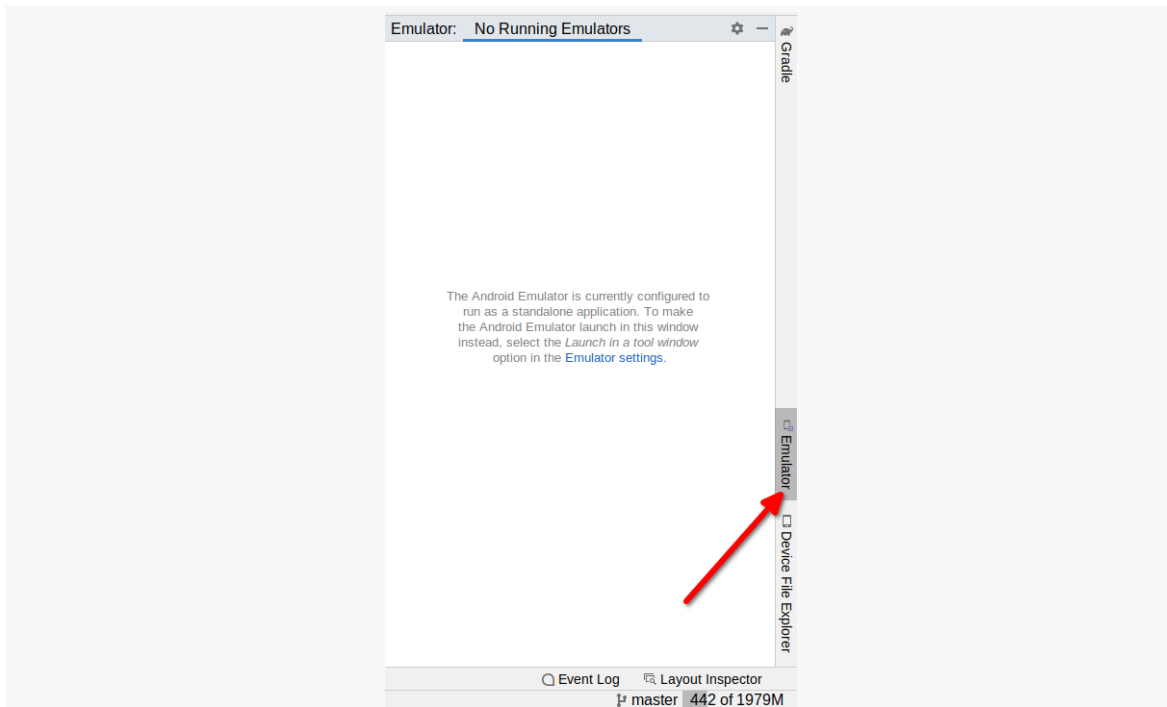


Figure 280: Emulator Tool Window, As Initially Opened, with Dock Item Highlighted

That message is:

The Android Emulator is currently configured to run as a standalone application. To make the Android Emulator launch in this window instead, select the *Launch in a tool window* option in the Emulator settings.

USING THE AVD MANAGER AND THE EMULATOR

“Emulator settings” is a link, leading you to the appropriate spot in the Settings dialog:

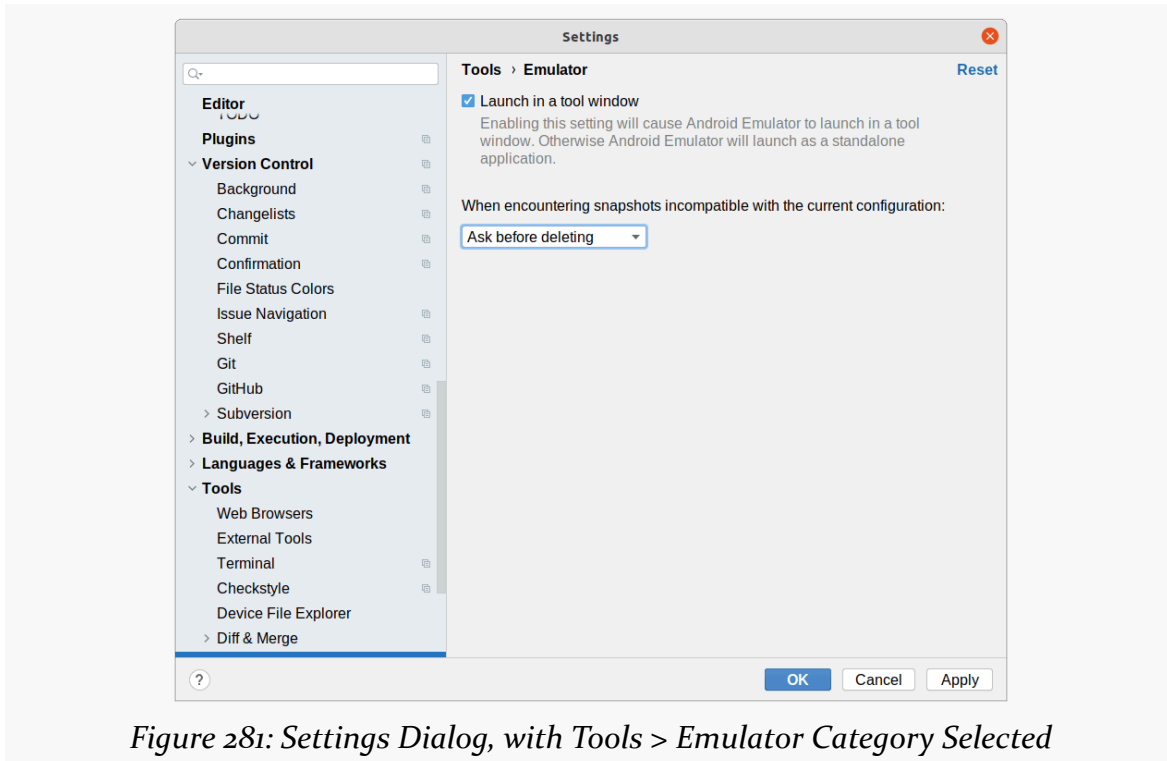


Figure 281: Settings Dialog, with Tools > Emulator Category Selected

Checking that checkbox and closing the dialog simply changes that message to:

No emulators are currently running. To launch an emulator, use the AVD Manager or run your app while targeting a virtual device.

USING THE AVD MANAGER AND THE EMULATOR

Now, if you run your app and choose an emulator as the launch target, rather than a separate window opening up, you get the emulator directly in this tool window, as we saw earlier in this section:

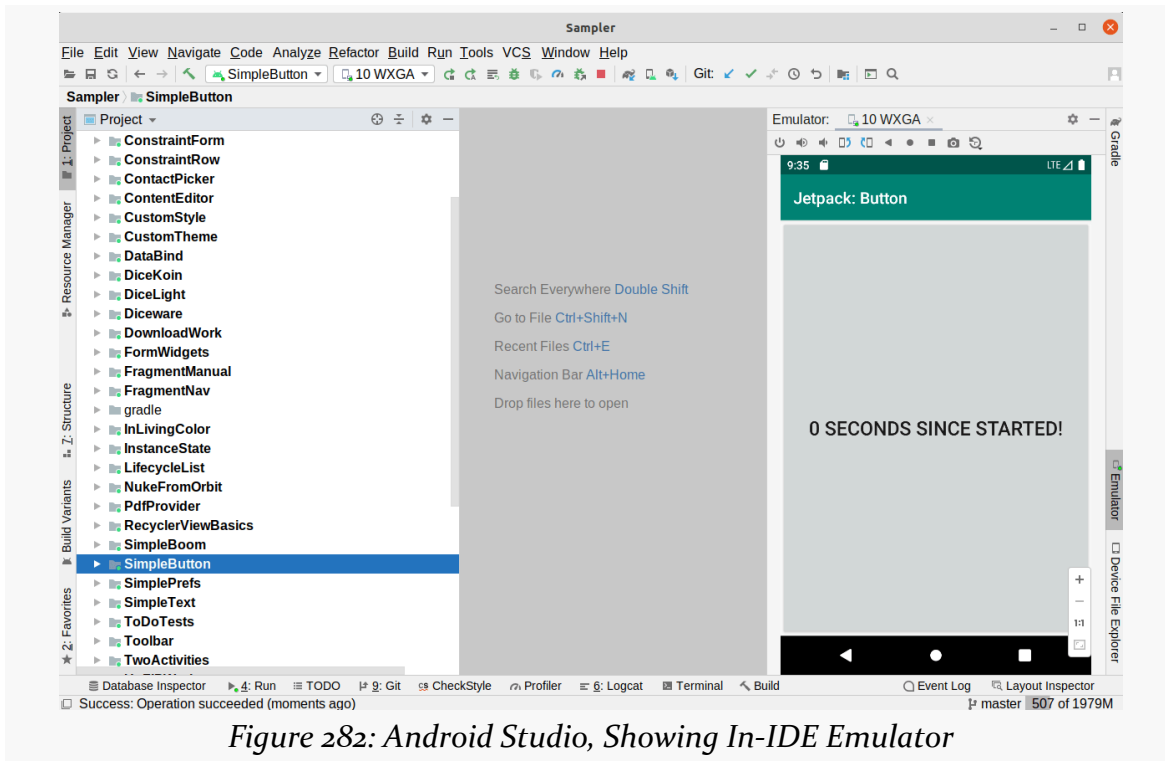


Figure 282: Android Studio, Showing In-IDE Emulator

The zoom controls towards the lower right will let you zoom in and out of the emulator screen. And there is a toolbar across the top that allows you to stop the emulator, control the volume, rotate the screen, record a screencast, or capture a screenshot.

USING THE AVD MANAGER AND THE EMULATOR

If you launch multiple emulators, they will show up as tabs, to toggle between them:

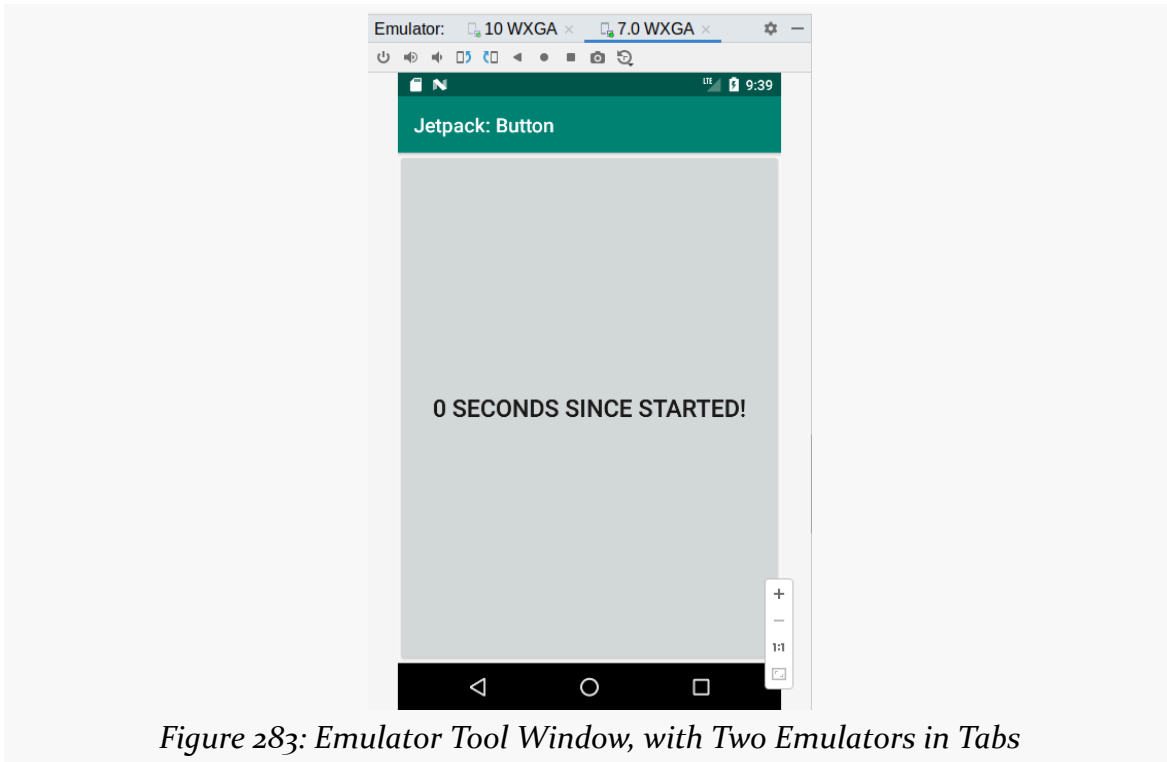


Figure 283: Emulator Tool Window, with Two Emulators in Tabs

However, as that original message indicated, you do not have access to the extended controls when running in this mode. For that, you would need to go back into Settings, disable this option from the Tools > Emulator category, and run the emulator normally. With luck, in a future Android Studio update, there will be a way to get to the extended controls panel from the in-IDE emulator.

Using the SDK Manager

When you installed Android Studio, along with it came some initial pieces of the Android SDK. That is enough to get you going, but eventually, you will need to download more of that SDK. That is handled through an “SDK Manager” portion of the Settings UI in Android Studio.

You can get to the SDK Manager in three ways:

- Choosing Tools > SDK Manager from the main menu
- Clicking the associated toolbar button:

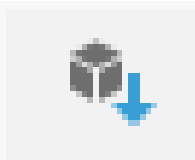


Figure 284: Android Studio SDK Manager Toolbar Button

- Choosing File > Settings from the main menu, then navigating to “Appearance & Behavior” > “System Settings” > “Android SDK” in the category tree on the left (or search for “Android SDK” in the search field)

Installing Platform Pieces

The SDK Manager has three tabs. The left-most of these is “SDK Platforms”, and it is where you can install things that are tied to specific Android OS versions:

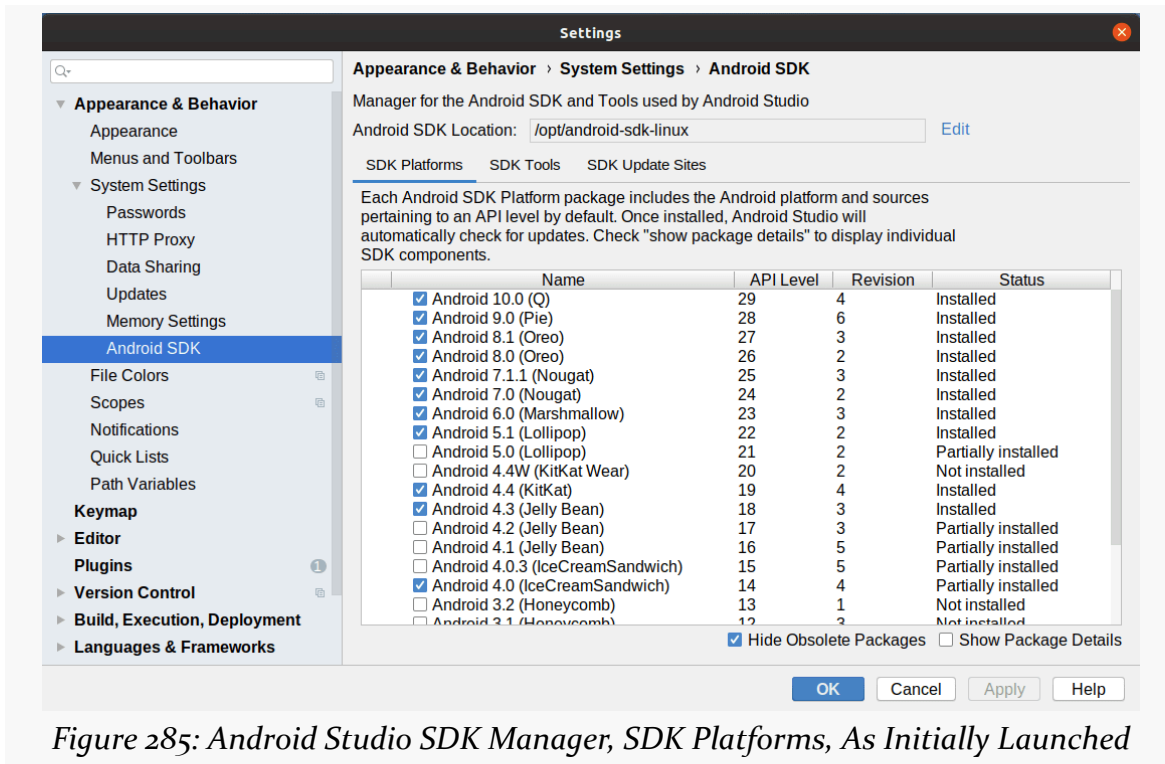


Figure 285: Android Studio SDK Manager, SDK Platforms, As Initially Launched

USING THE SDK MANAGER

The table lists Android OS versions along with a status of “Installed”, “Partially installed”, or “Not installed”. Those statuses are not especially accurate — to get a better understanding of what you have and what you can get, check the “Show Package Details” checkbox towards the bottom of the dialog. This turns the table into a tree-table, showing more precise information:

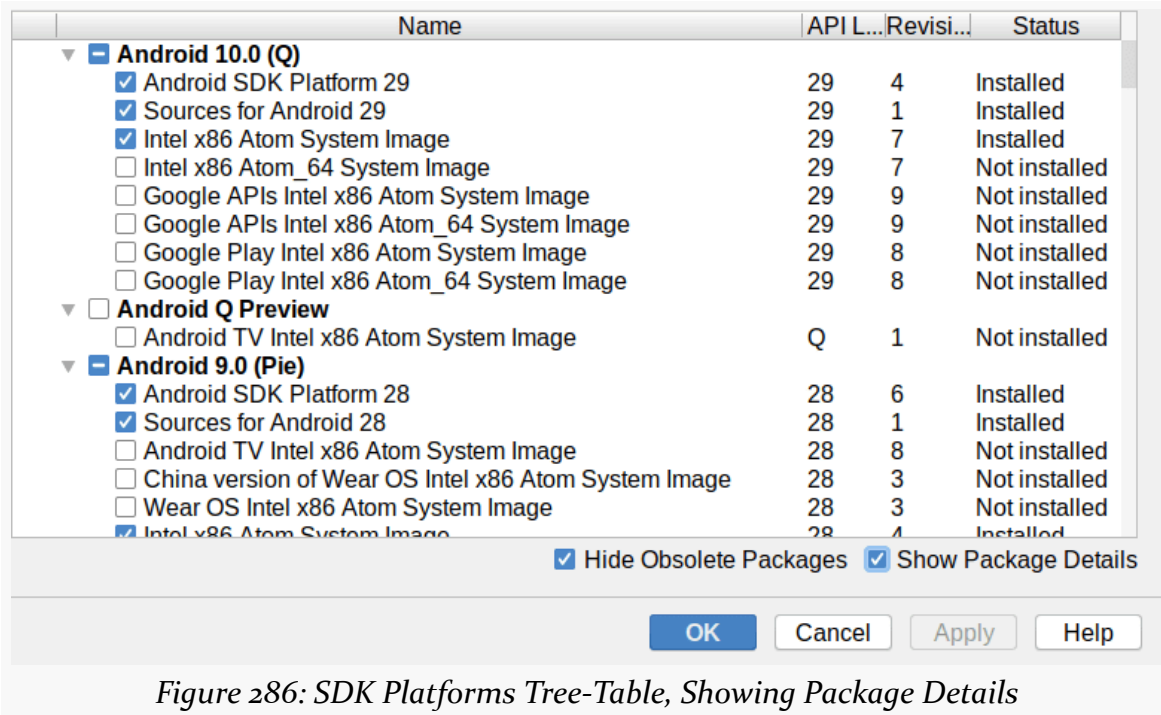


Figure 286: SDK Platforms Tree-Table, Showing Package Details

In general:

- “Android SDK Platform NN” (for some API level value NN, like 29), represents the stuff needed to allow you to have that version specified in `compileSdkVersion` in your module’s `build.gradle` file.
- “Sources for Android NN” is the source code for the Android SDK for that API level. This allows you to do things like step into the source code in the debugger, or to jump to the declaration of an SDK-supplied class or method in the IDE.
- “... System Image” represent emulator images that you can use as the basis for an AVD.

If you wish to change what you have installed, check or uncheck the various items, then click either “Apply” (to make the changes) or “OK” (to make the changes and close the dialog).

USING THE SDK MANAGER

A few times per year, you may find yourself in this tab, downloading SDK bits for a new Android version (and perhaps uninstalling older bits to free up disk space).

Installing and Upgrading Tools

The second tab is “SDK Tools”. This contains libraries, tools, and other elements of the Android SDK that are not strictly tied to API level:

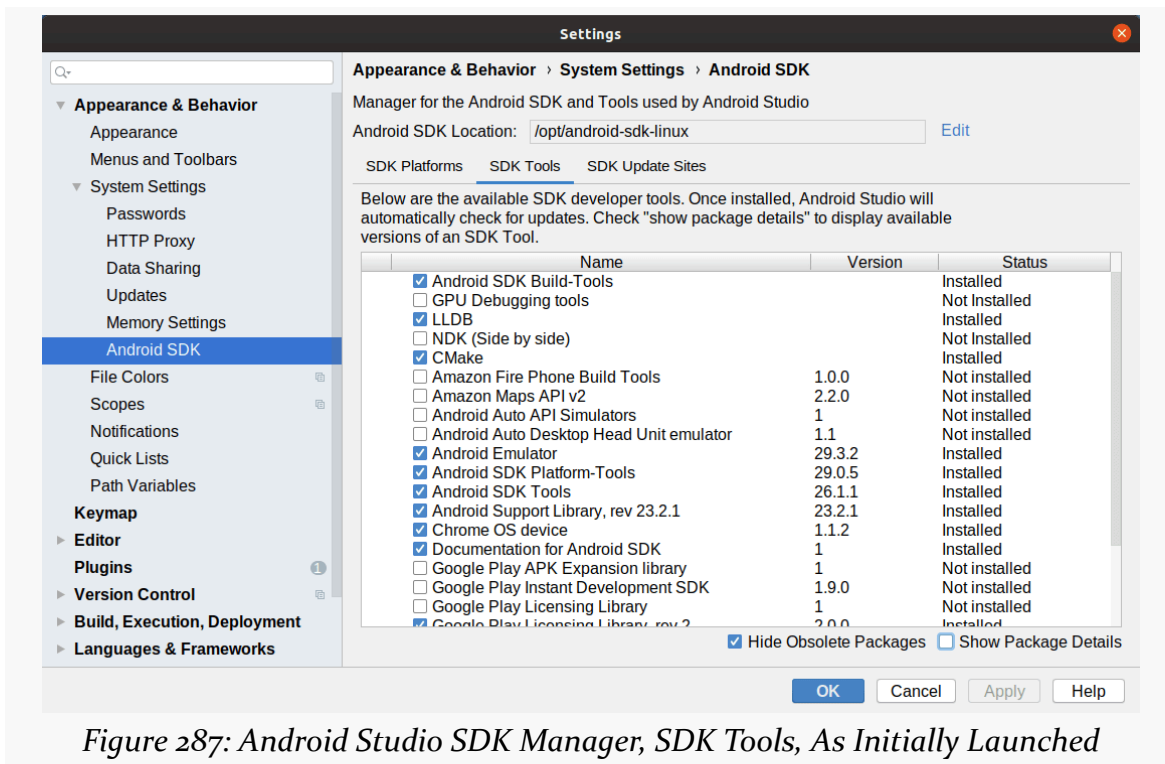


Figure 287: Android Studio SDK Manager, SDK Tools, As Initially Launched

Usually, unless some documentation or in-IDE instructions tells you to change something in here, you can leave it alone.

Adding Third-Party SDK Suppliers

Some third parties distribute their own SDK tools and related items through Android's SDK Manager. They may advise you to visit the third tab, "SDK Update Sites", and add their sites through it:

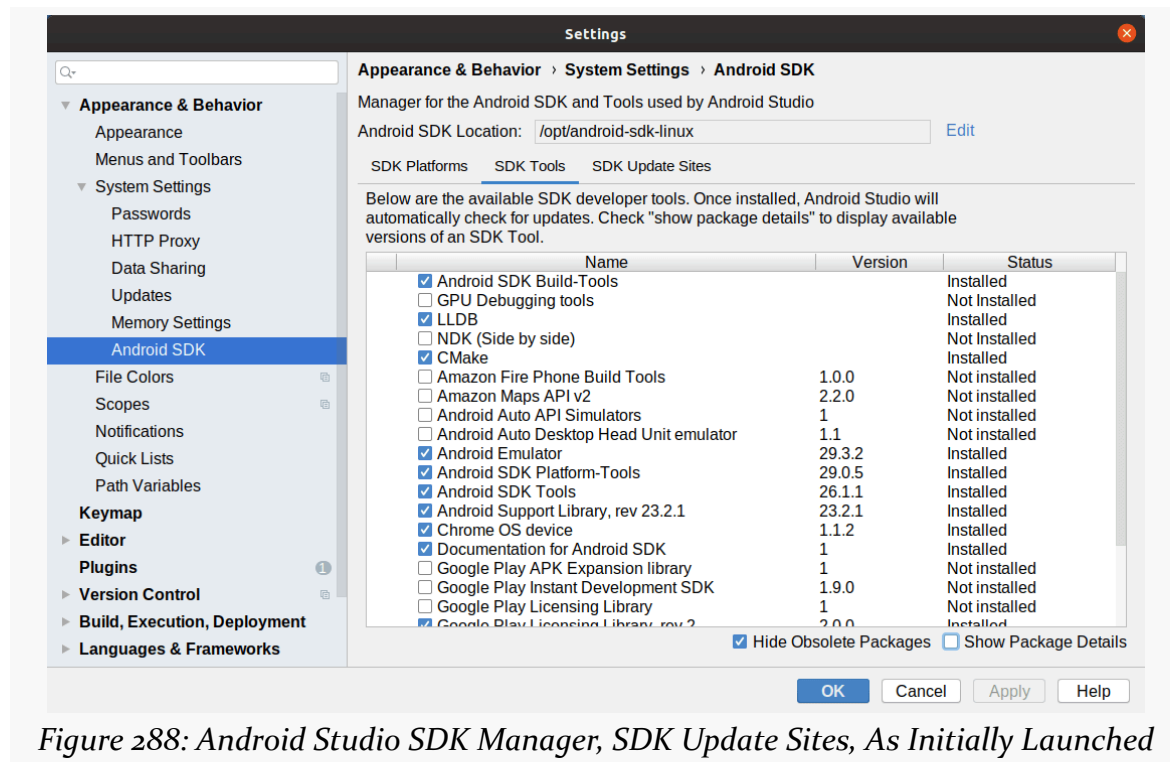


Figure 288: Android Studio SDK Manager, SDK Update Sites, As Initially Launched

The "+" and "-" items in the toolstrip on the side of the table allow you to add and remove rows, while the checkboxes control whether a given SDK source is enabled or not.

Changing items in this tab is rather unusual. Only change things here if directed to do so, and then ideally only if you trust the supplier of the SDK.

Configuring Your Project

In many of the samples in the book, we saw various settings in `build.gradle` files:

- At the top level of the `Sampler` or `SamplerJ` projects
- In individual modules, such as `Bookmarker`

The focus has been on where various settings reside in those files (e.g., stuff that goes in the `dependencies` closure, stuff that goes in the `android` closure).

However, Android Studio offers another way of manipulating those settings, via the Project Structure dialog. You can open this from the “File” > “Project Structure” main menu option, and it lets you manipulate many of the same things that we covered, but using a GUI rather than a text editor.

Risks and Rewards

So... why did we bother with all that `build.gradle` stuff, if we could use a GUI?

Mostly, it is because not everything can be configured via the Project Structure dialog. You cannot [enable data binding](#), for example. Nor can you add Gradle plugins, like [the “safe args” plugin for the Navigation component](#). Once you get past some fairly basic settings, the Project Structure dialog cannot help you. So, understanding how `build.gradle` files are set up is important, for those things that cannot be manipulated via the Project Structure dialog.

Also, if your `build.gradle` file does contain things that cannot be managed by the Project Structure dialog, there is a risk that using the Project Structure dialog will somehow break those “things”. Bear in mind that a Gradle build file is a Groovy script — `build.gradle` is a program that builds an object model with the details of

how to build your app. While we have a lot of code generators, in Android and beyond, usually a code generator is a write-only tool. Using other data as input, the code generator writes out code. Here, though, the Project Structure dialog needs to *modify* code that is otherwise maintained by hand. That is a complex programming problem, and bugs in the Project Structure dialog might overwrite or otherwise mess up some of that hand-entered Groovy code.

Google obviously feels fairly confident in the Project Structure dialog, and you are welcome to use it. This chapter will outline what sorts of things you can configure using that dialog. Just remember that anything configurable in this dialog can be configured in by hand, typically in a `build.gradle` file. And, be sure to keep good backups or version control history, to be able to recover if Project Structure breaks the build.

The Project Category

When you open the Project Structure dialog from the “File” > “Project Structure” main menu option, you will see a series of categories down the left side. Clicking on those will change the form available on the right side.

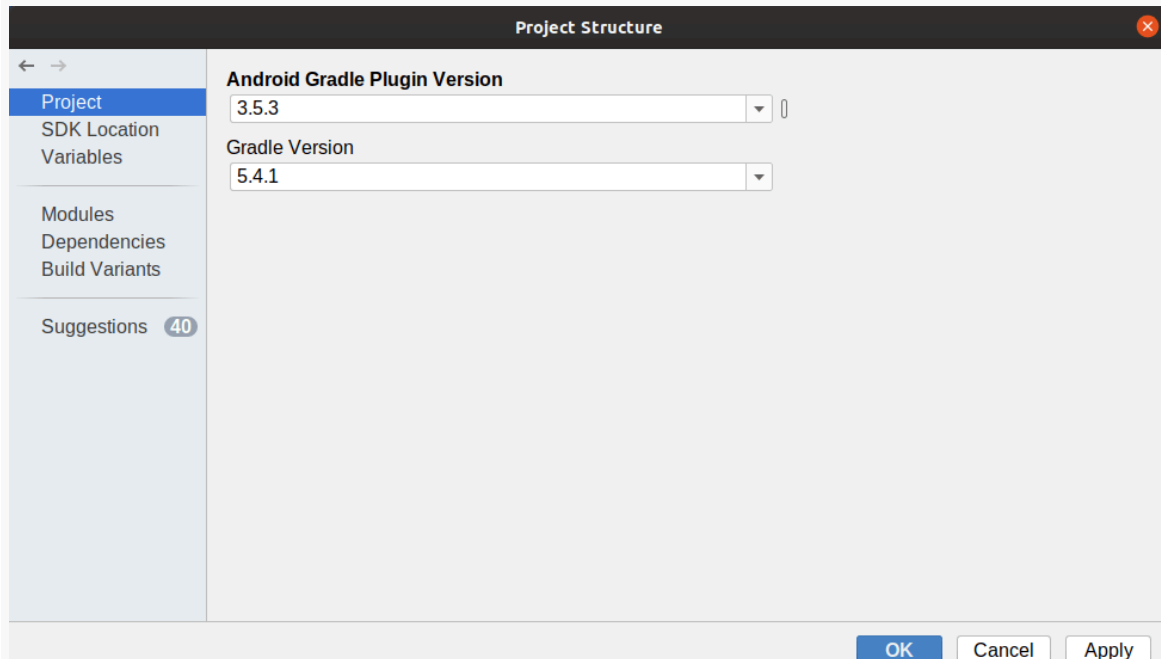


Figure 289: Project Structure Dialog, Project Category

The first of those categories is the “Project” category. Here, you can configure two things:

- What version of Gradle to use, which affects the value of the `distributionUrl` in `gradle/wrapper/gradle-wrapper.properties`
- What version of the Android Gradle Plugin to use, which controls the `com.android.tools.build:gradle` version in the `classpath` entry in your `buildscript dependencies` closure in the top-level `build.gradle` file

The SDK Location

The “SDK Location” category supplies paths to where the Android SDK, Android NDK (for C/C++ development), and Java JDK reside on your development machine:

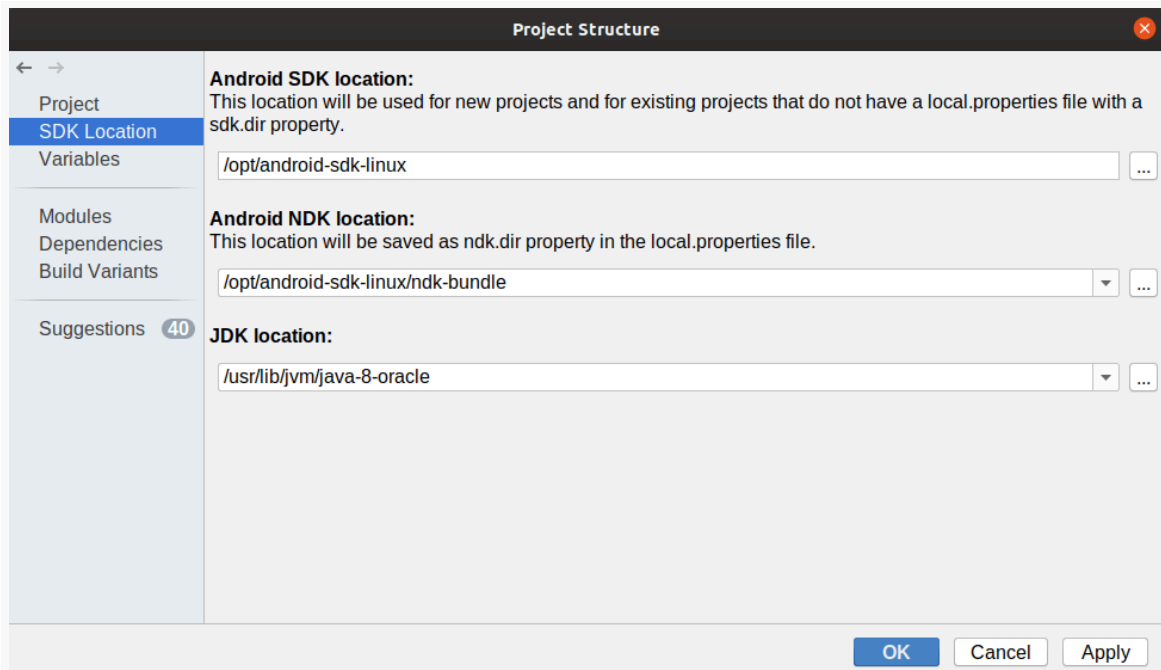


Figure 290: Project Structure Dialog, SDK Location Category

For many Android Studio installations, these will point where Android Studio set up these development kits. Developers who installed one or more of these manually (e.g., JDK via a Linux package manager) will have less-typical locations reflected in these fields.

The Variables

In our Kotlin files, we sometimes define constants, such as the ext constants in the top-level build.gradle file:

```
buildscript {
    ext.nav_version = "2.3.5"

    repositories {
        google()
        jcenter()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:7.0.2'
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
    }
}
```

(from [build.gradle](#))

You can also attempt to manipulate these from the “Variables” category of the Project Structure dialog:

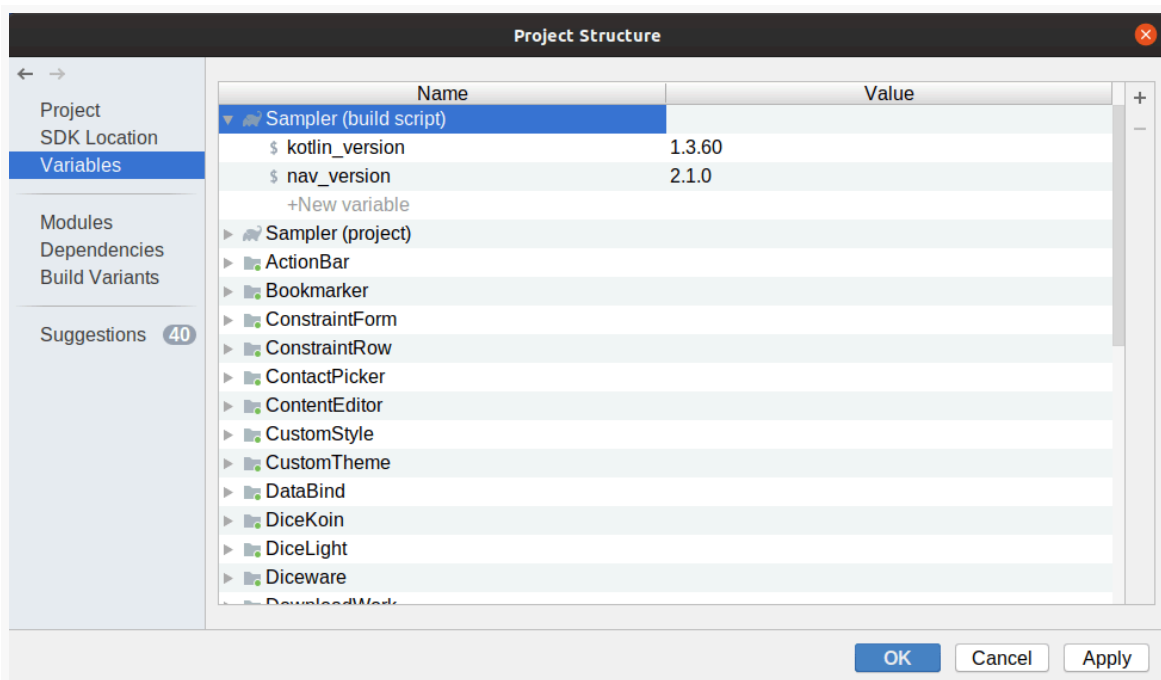


Figure 291: Project Structure Dialog, Variables Category

CONFIGURING YOUR PROJECT

Here, we see the `kotlin_version` and `nav_version` constants that we defined in the top-level `build.gradle` file. The fields to the right of the constant name are editable, so you can use those to change the constants' values.

In principle, you can use the “+New variable” options to define new variables. In the opinion of this author, this feature does not work very well.

Also, on the toolbar on the right, there is a “-” toolbar button that is enabled when you have selected a variable. Clicking this button will delete the selected variable, after a confirmation dialog. Just be careful: the IDE does not validate whether you are still using that variable!

The Modules

The “Modules” category lists all of the modules in your project and allows you to configure them. In many projects, there will simply be an app module and nothing else. In the book's sample projects, there are more modules than normal:

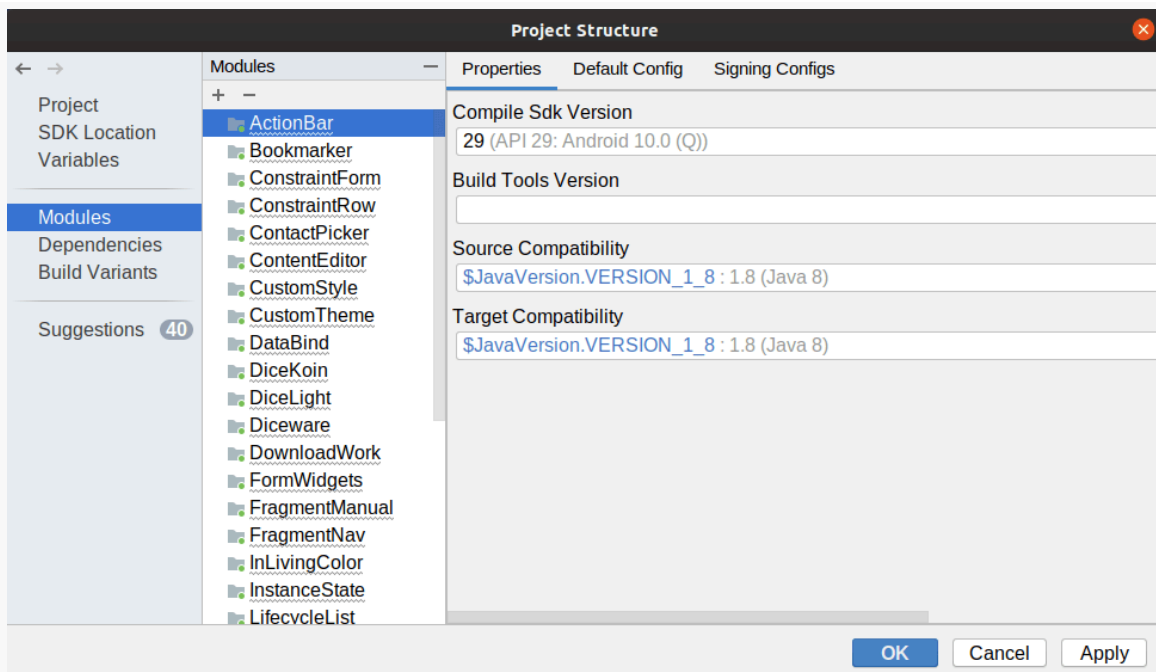


Figure 292: Project Structure Dialog, Modules Category, Properties Tab

The gray undersquiggle underneath the module names means that there is a

CONFIGURING YOUR PROJECT

suggestion for changes to the module's configuration. We will explore those suggestions more [later in the chapter](#).

Module Properties

There are three tabs in the “Modules” category. The first one is “Properties”, which allows you configure items from the selected module's `build.gradle` file, such as:

- The `compileSdkVersion`
- The version of the `dx` compiler, `aapt` packager, and related tools that you use (`buildToolsVersion`, typically left unspecified)
- What version of Java to support, in terms of how that Java gets compiled (the `sourceCompatibility` and `targetCompatibility` values in the `compileOptions` closure)

Default Config

The “Default Config” tab lets you configure a variety of items that appear in the `defaultConfig` closure of the selected module's `build.gradle` file:

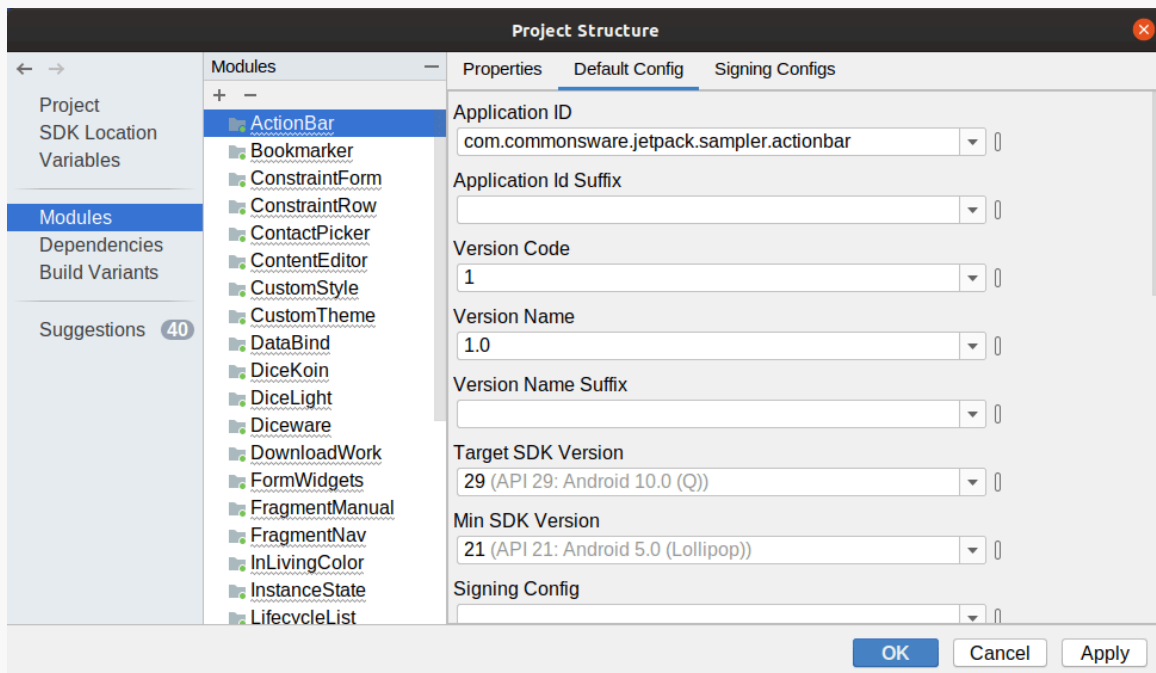


Figure 293: Project Structure Dialog, Modules Category, Default Config Tab

CONFIGURING YOUR PROJECT

Many of these are beyond the scope of this book. Some of the ones that we have covered are:

- The `applicationId`
- The `versionCode` and `versionName`
- The `minSdkVersion` and `targetSdkVersion`

Signing Configs

In [the chapter on signing your app](#), we saw how to use Android Studio to sign your APK. There is another option: configuring Gradle to sign your APK. This is mostly used in cases where the APK will be generated and signed automatically, such as via a build server.

The “Signing Configs” tab contains a form for you to teach Gradle how to sign your APK:

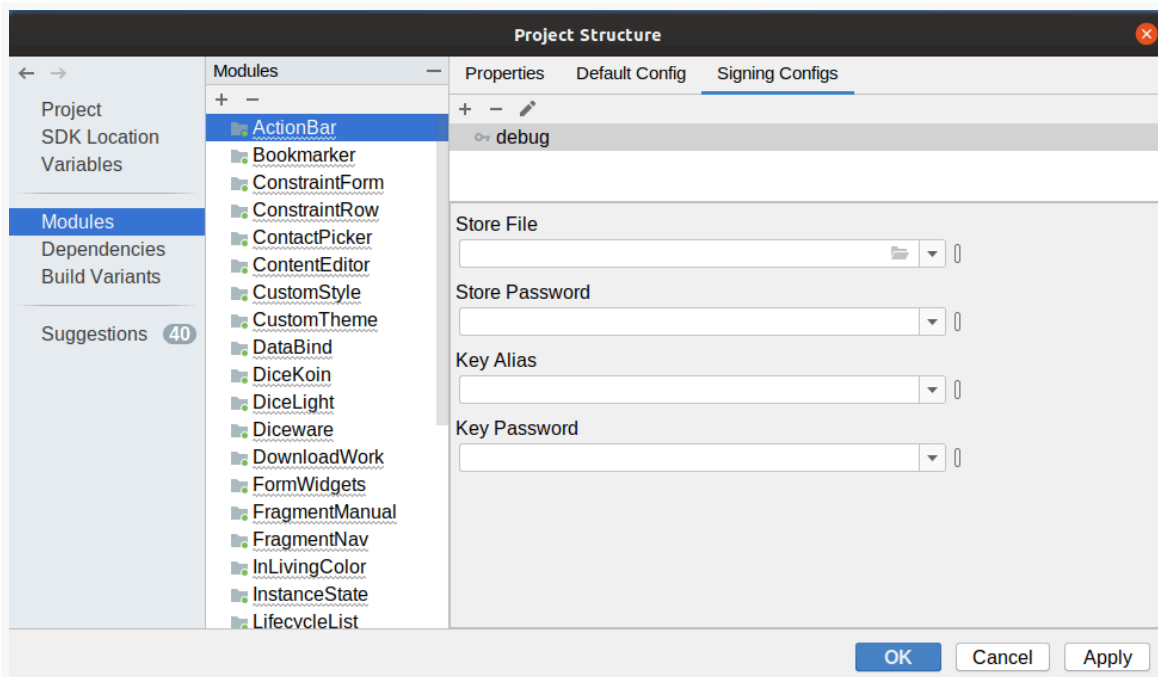


Figure 294: Project Structure Dialog, Modules Category, Signing Configs Tab

The details of how this works, and the resulting Gradle constructs, can be found [in the Android SDK documentation](#).

Dependencies

The “Dependencies” category lets you maintain the dependencies of your modules... to an extent:

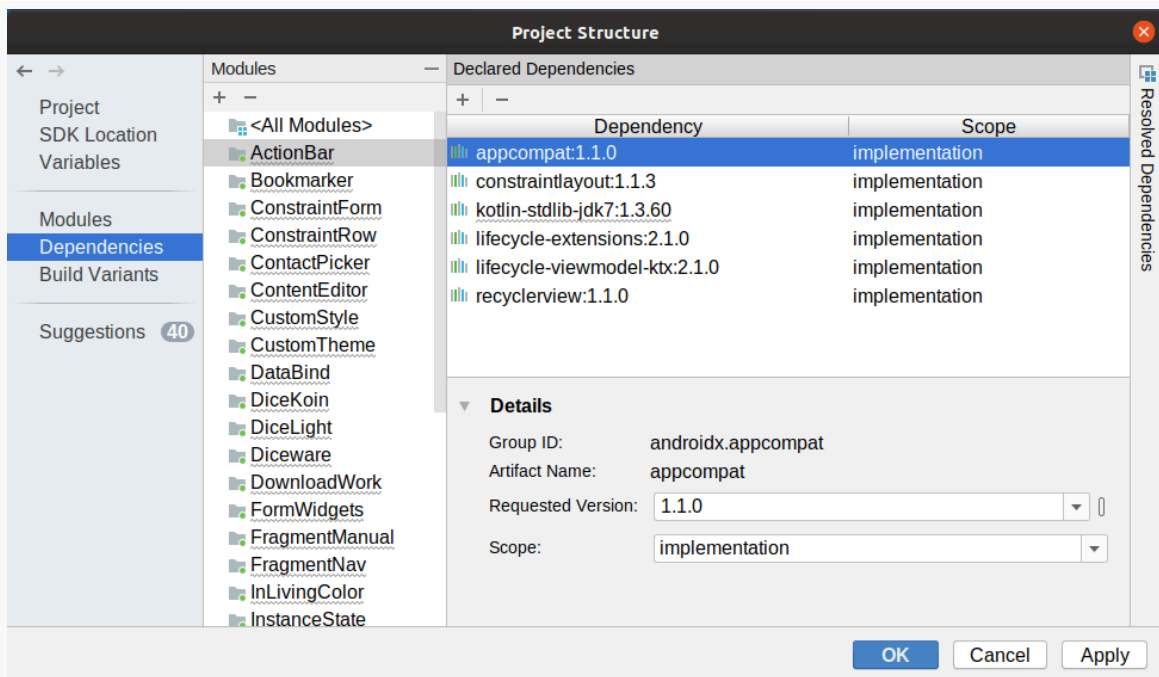


Figure 295: Project Structure Dialog, Dependencies Category

It shows you the current dependencies listed for a given module. Using the “Details” form below the list, you can change the version number and dependency type (e.g., implementation versus testImplementation).

CONFIGURING YOUR PROJECT

The “Resolved Dependencies” tool docked on the right will show you the full set of dependencies for your module, including transitive dependencies:

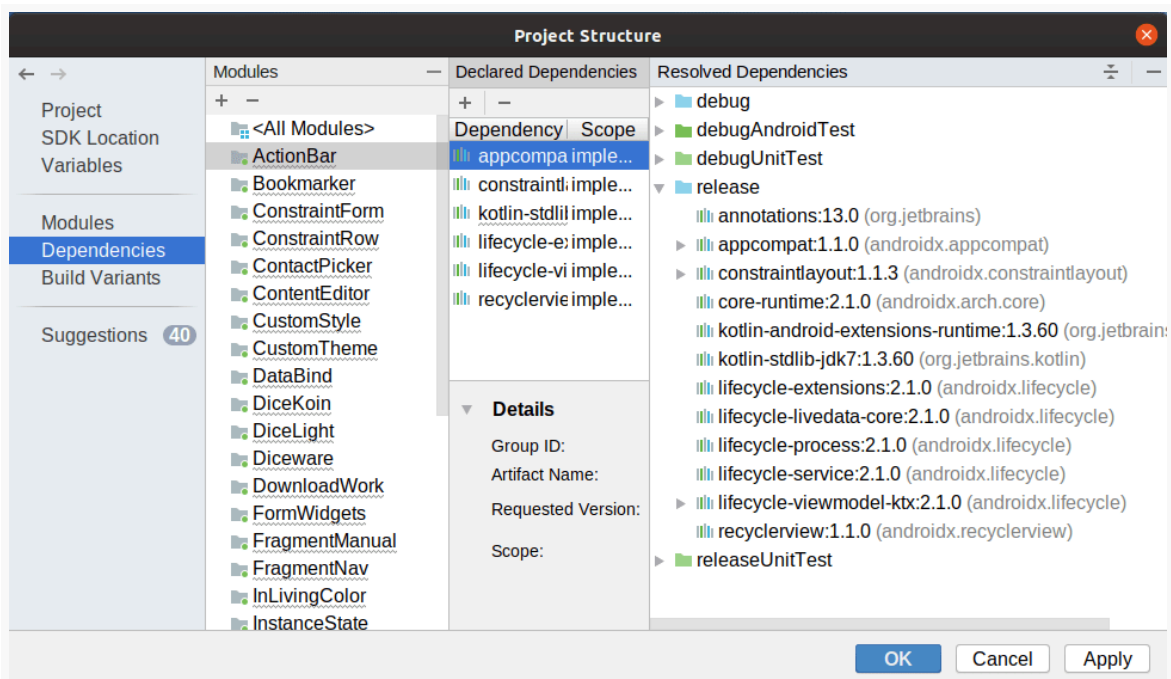


Figure 296: Project Structure Dialog, Dependencies Category, Showing Resolved Dependencies

The tree structure lets you examine scenarios (e.g., `debugUnitTest`, `release`) and the dependencies for each.

The “-” button above the list lets you remove the dependency selected in the list. The “+” button adjacent to it theoretically lets you add a dependency, but this does not work very well.

Build Variants

In our `build.gradle` file for a module, we sometimes define different settings for different scenarios. For example, many of our modules have had a `buildTypes` closure, which can provide different configuration options for debug builds (the ones that we normally run) and release builds (the ones that users run):

CONFIGURING YOUR PROJECT

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
```

(from [ActionBar/build.gradle](#))

The “Build Variants” category lets you configure those build types to an extent:

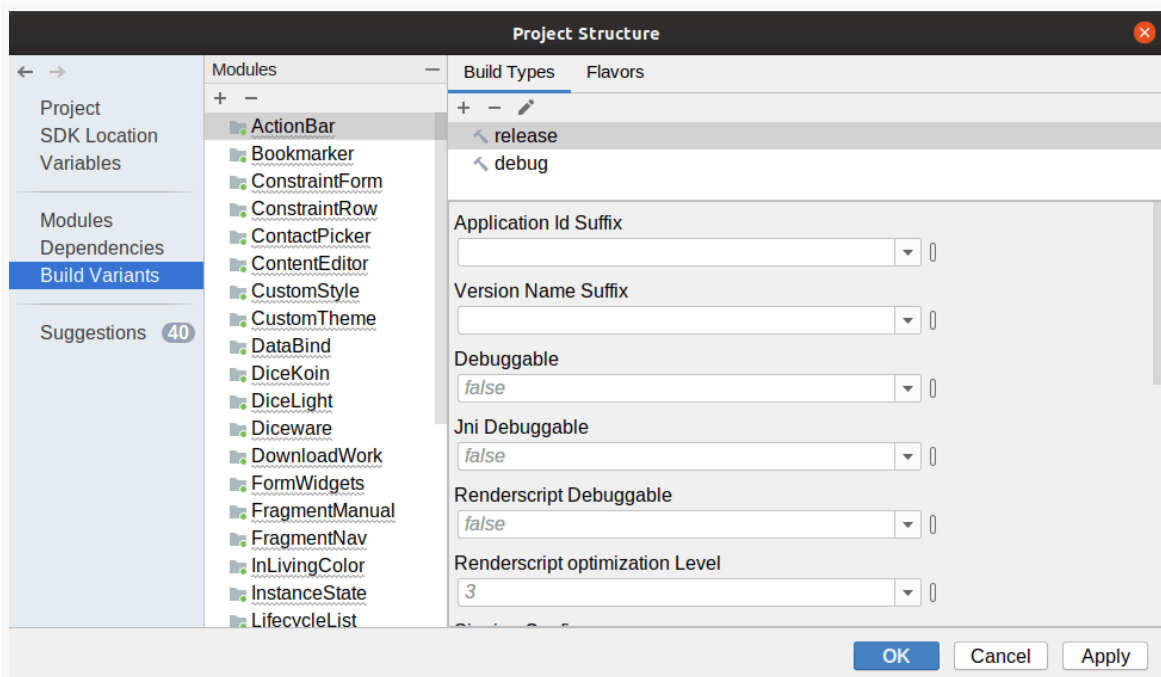


Figure 297: Project Structure Dialog, Build Variants Category

The “Flavors” tab is for [product flavors](#), an advanced technique for having different variants of your app for different distribution scenarios (e.g., free vs. paid, Google Play Store vs. another distribution site).

Suggestions

The “Suggestions” tab, as the name indicates, offers suggestions for changes that may be appropriate to your modules:

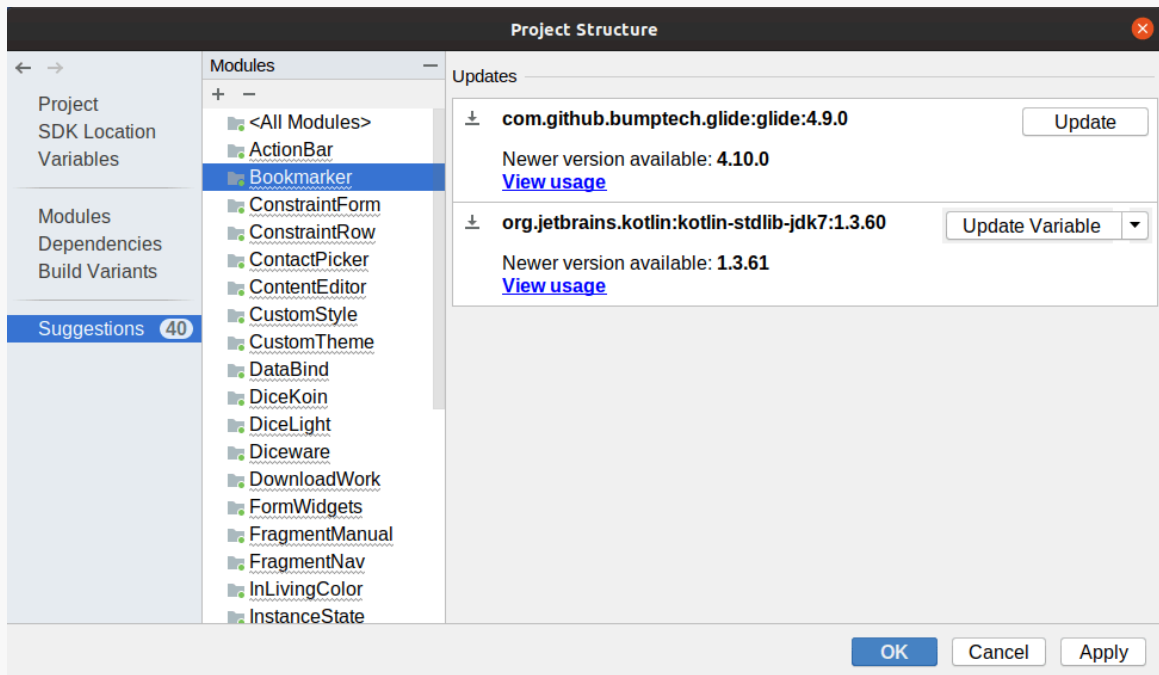


Figure 298: Project Structure Dialog, Suggestions Category

Mostly, it is focused on out of date dependencies. For the selected module, the suggestions will consist of a set of cards with details of the suggestion and (often) a button to make the suggested change.

Configuring Android Studio

There are many things that you can configure about the operation of Android Studio. Most of those are in the Settings dialog, available from the “File” menu on Windows and Linux and via “Android Studio” > “Preferences...” on macOS.

In there, you will find a category tree on the left with over 100 different groups of settings that you can configure.

We will not look at all of them here. You would get bored.

Searching for Settings

The search field at the top of the category tree can help you find settings buried in the overall set of Settings:

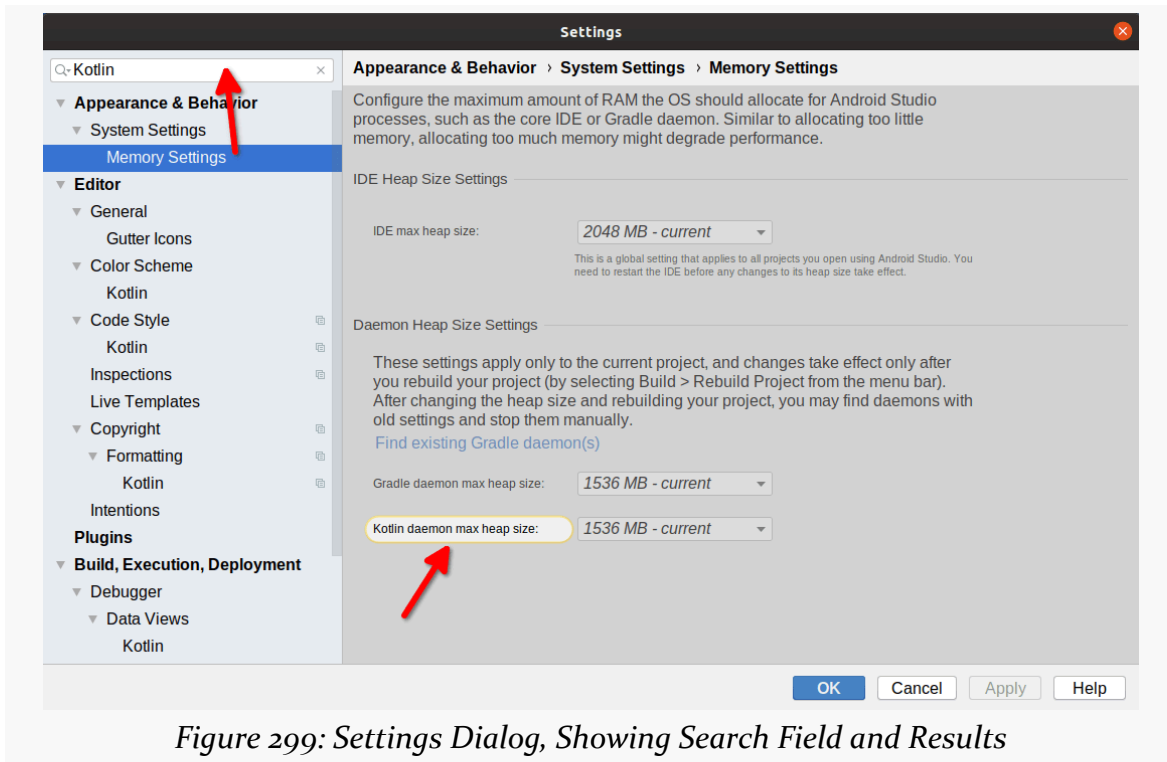


Figure 299: Settings Dialog, Showing Search Field and Results

For many of the categories that have settings that refer to your searched-for keyword, the particular setting(s) will be highlighted, to help draw your eye to the relevant values.

Themes and Colors

Some developers like light-colored IDEs. Others prefer dark-colored IDEs. When you install Android Studio, you typically get a choice of a theme, and you can change that choice later on in the “Appearance & Behavior” > “Appearance” category, via the “Theme” drop-down:

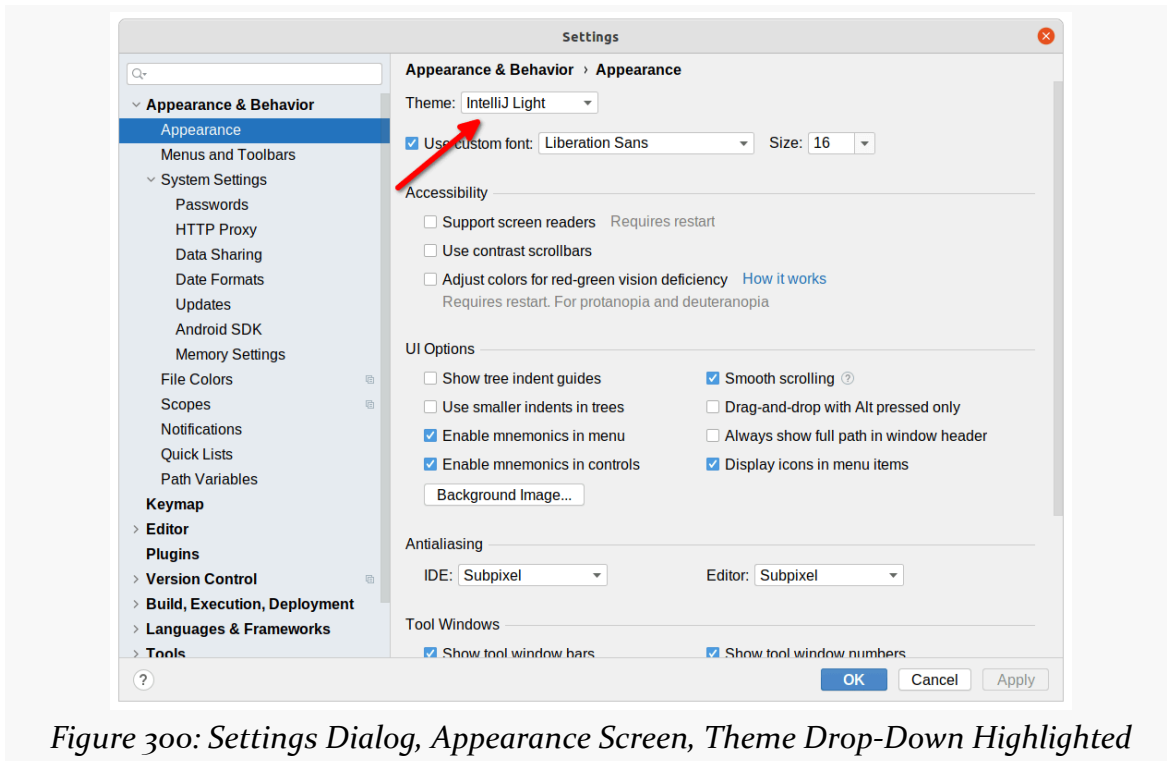


Figure 300: Settings Dialog, Appearance Screen, Theme Drop-Down Highlighted

The stock light-colored theme is “IntelliJ”, named after the IntelliJ IDEA tool that Android Studio itself is based upon.

CONFIGURING ANDROID STUDIO

The stock dark-colored theme is “Darcula”, named after [a fictional character](#) with a couple of letters transposed in the name:

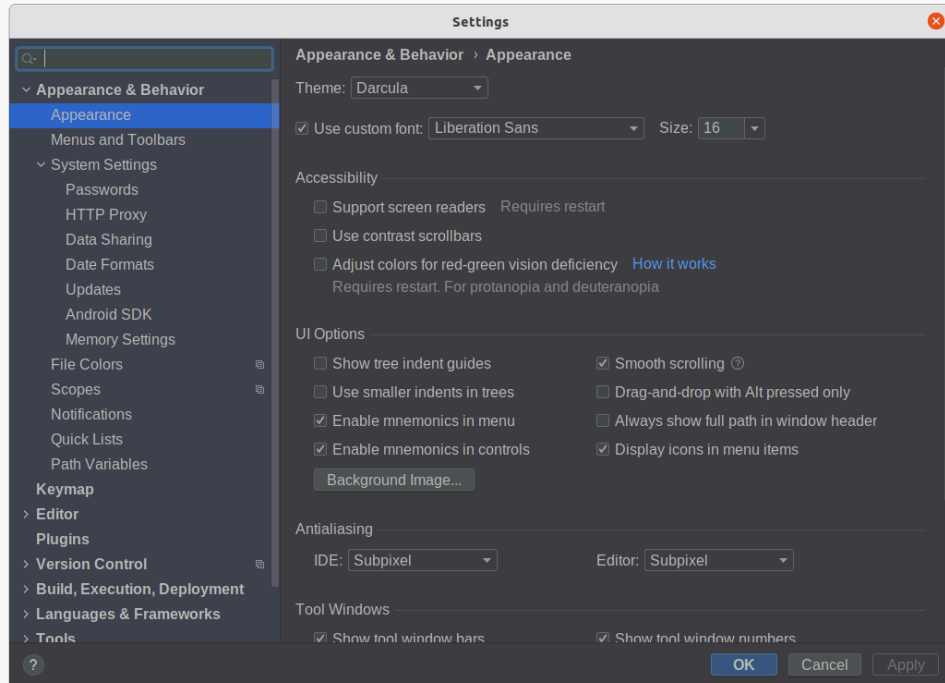


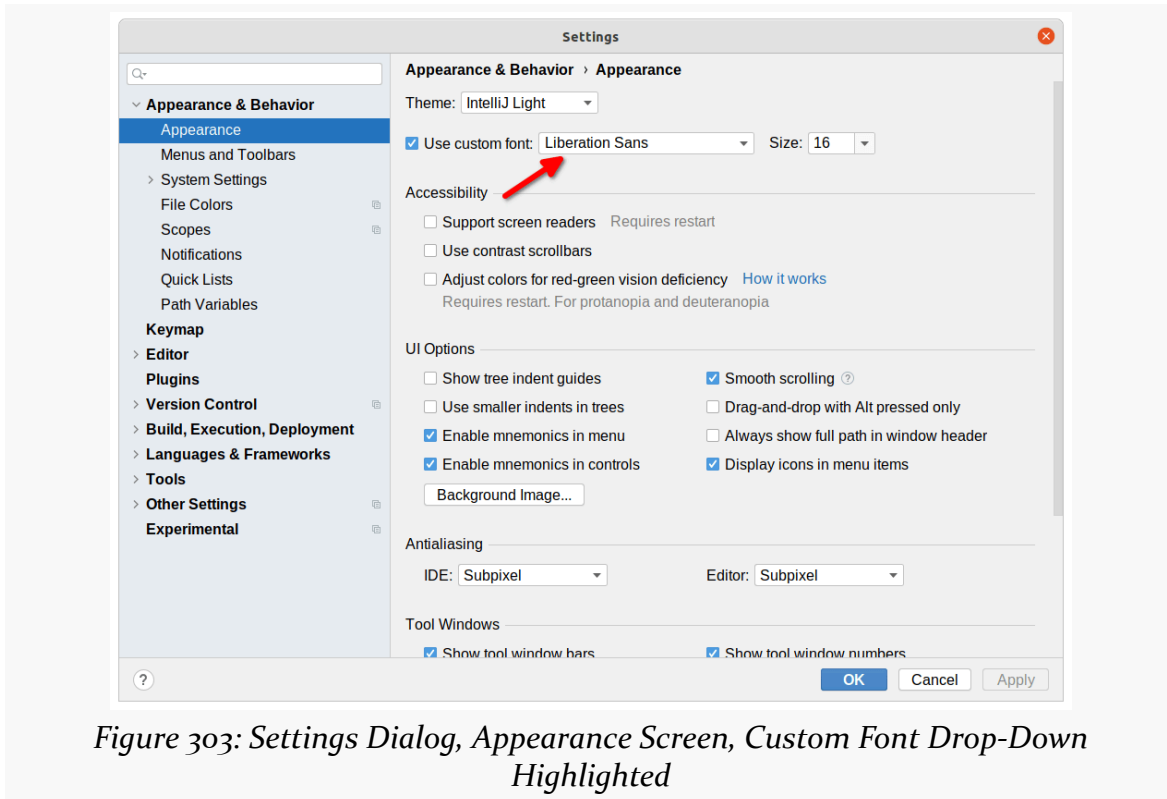
Figure 301: Settings Dialog, Appearance Screen, in Darcula Theme



After the color scheme, perhaps the thing that developers like to customize the most in an IDE is the font. Android Studio lets you do that... but there are a few different fonts in play.

CONFIGURING ANDROID STUDIO

Back up in the “Appearance & Behavior” > “Appearance” category, for example, there is a “Use custom font” option, just below the theme drop-down:



This controls the font used for menus, buttons, captions, and similar things. While you might want to customize that, the font that developers tend to be more concerned about is the font used in the editor... and that is determined elsewhere.

CONFIGURING ANDROID STUDIO

In the “Editor” > “Font” category, you can control the font used by the editor:

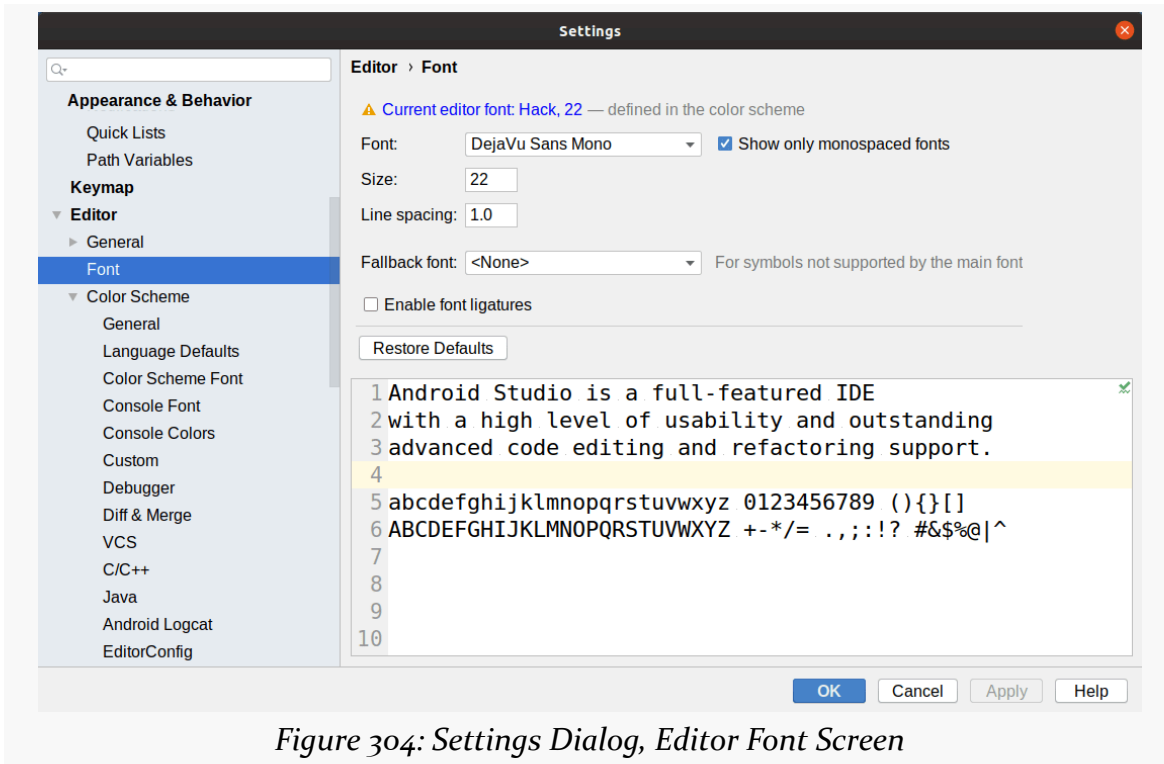
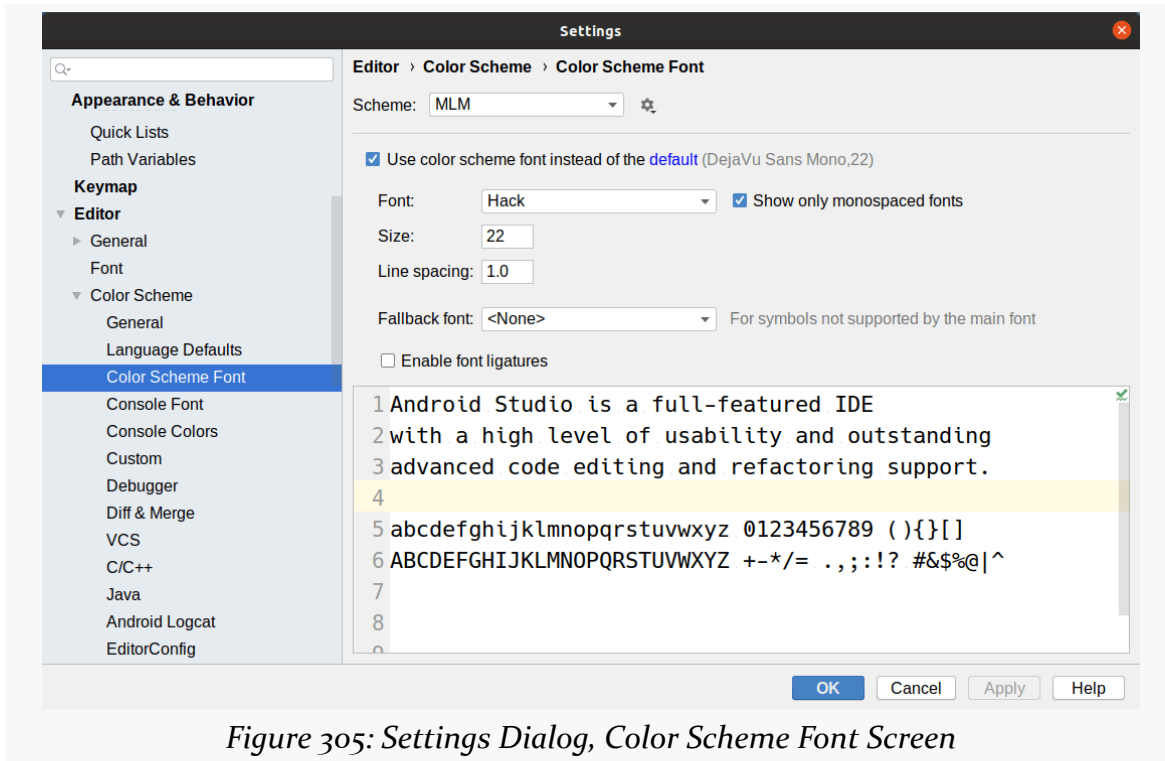


Figure 304: Settings Dialog, Editor Font Screen

CONFIGURING ANDROID STUDIO

This provides the default font used by the editor. However, as part of customizing a color scheme — as shown in the previous section — you can also customize the font there, in the “Editor” > “Color Scheme” > “Color Scheme Font” category:



So, for the editor, the font in the color scheme is used if one is defined there, otherwise the default editor font is used.

CONFIGURING ANDROID STUDIO

There are other miscellaneous fonts that you can configure as well, such as the default font used in console-style tools like the Terminal:

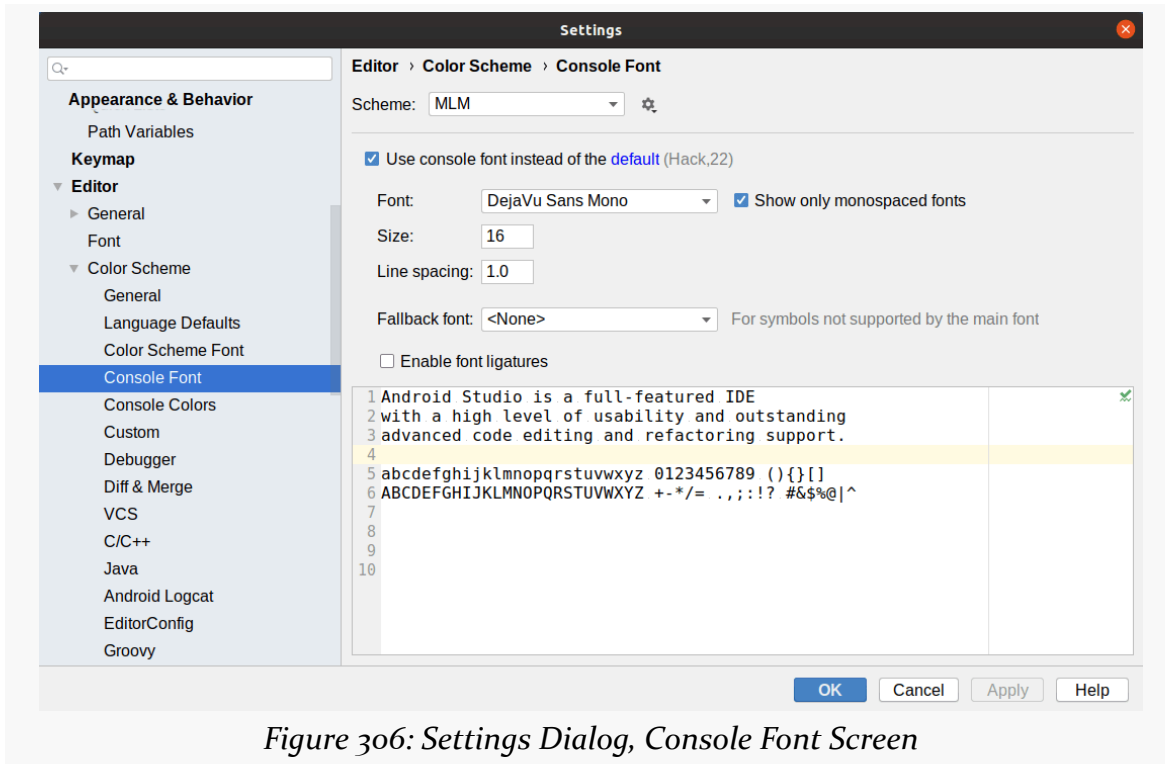


Figure 306: Settings Dialog, Console Font Screen

By default, your console font will be the same as your editor font.

Code Styles

For you as an individual developer, colors and fonts may be your most important settings.

If you are part of a development team, though, the code style might be the most critical. Code style handles everything from how big is a tab to whether wildcard imports (`import android.widget.*`) are allowed.

CONFIGURING ANDROID STUDIO

Android Studio offers per-language code style configuration, such as for Kotlin:

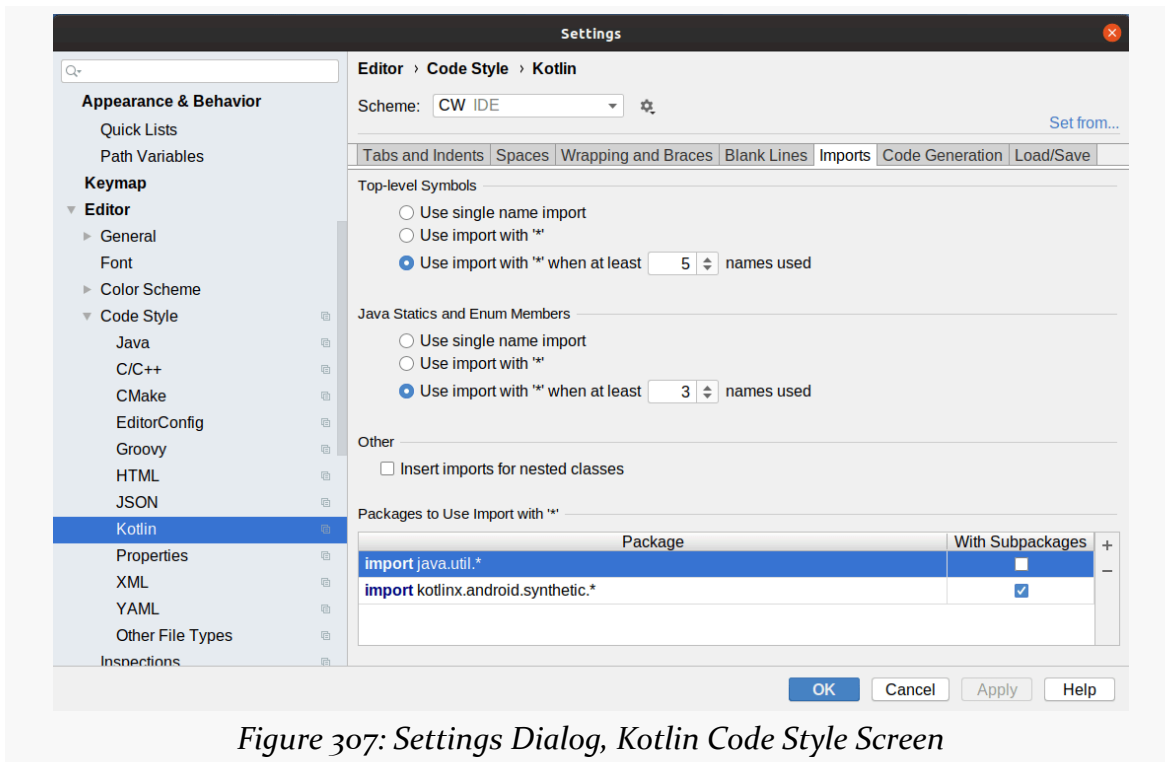


Figure 307: Settings Dialog, Kotlin Code Style Screen

CONFIGURING ANDROID STUDIO

As with color schemes, code styles are also part of a scheme. The IDE has a base set, and you use “Copy to Project” to have project-specific settings:

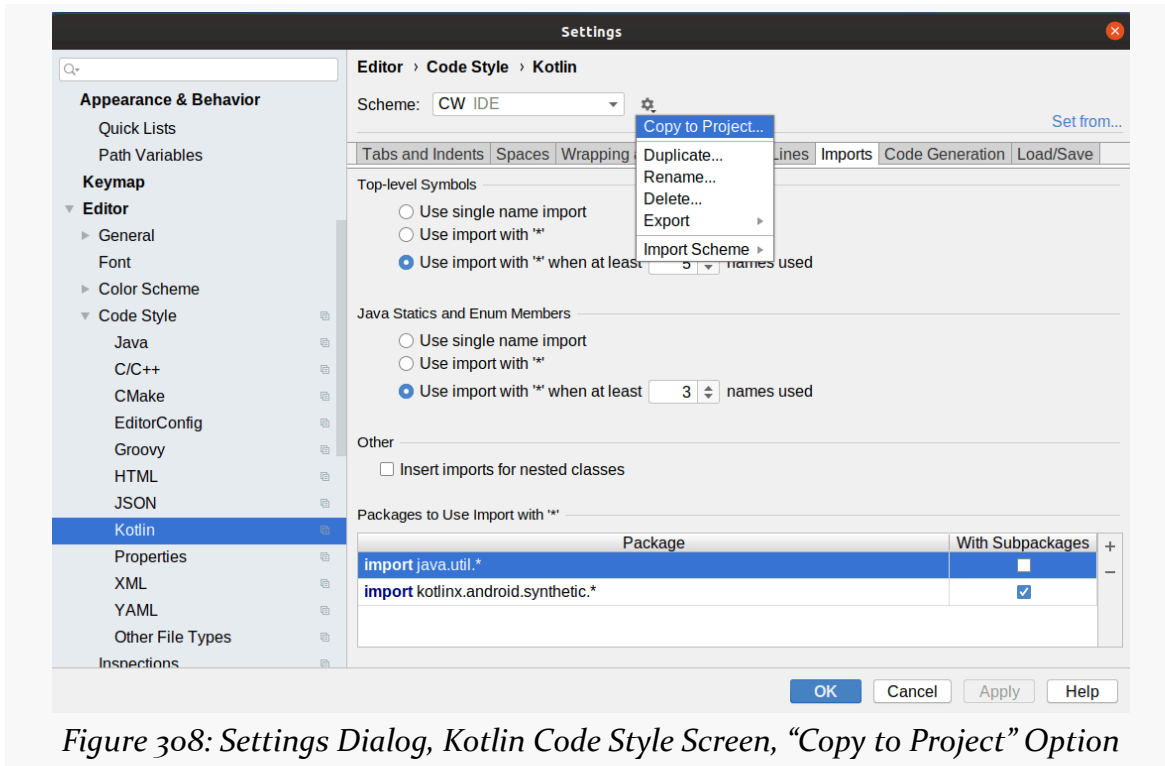


Figure 308: Settings Dialog, Kotlin Code Style Screen, “Copy to Project” Option

CONFIGURING ANDROID STUDIO

For Kotlin in particular, you might want to adopt the official Kotlin style guide, created by JetBrains when they created the Kotlin language. To use its settings as your starting point, you can click the “Set from...” link on the edge of the dialog:

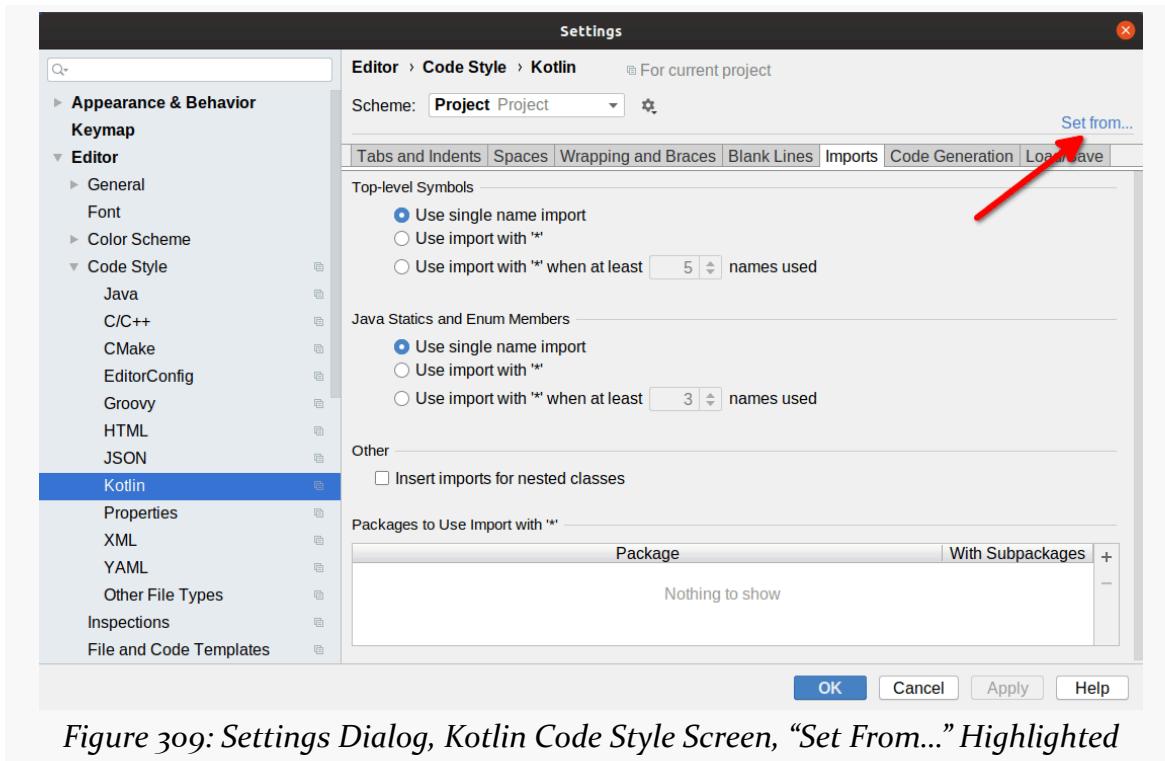


Figure 309: Settings Dialog, Kotlin Code Style Screen, “Set From...” Highlighted

Clicking that will display a fly-out menu tree, and in there is a “Kotlin style guide” option. Choosing that will update the settings across the various tabs to match the Kotlin style guide.

CONFIGURING ANDROID STUDIO

However, the Kotlin style guide does not specify everything. For example, it does not state the maximum length of a line of code. If you want to share settings for those among team members, you can enable EditorConfig support:

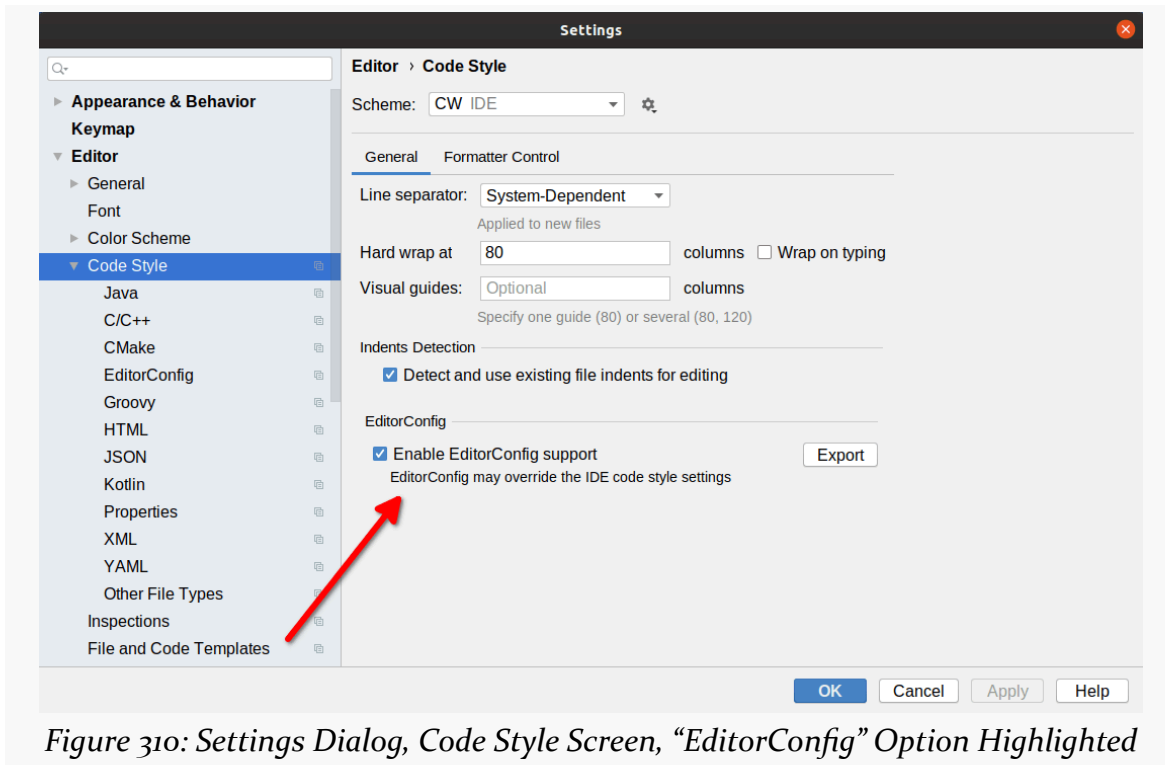


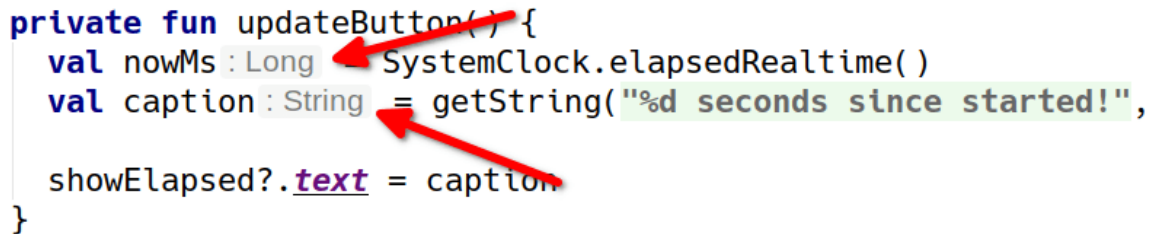
Figure 310: Settings Dialog, Code Style Screen, “EditorConfig” Option Highlighted

Then, if your project has an `.editorconfig` file in the project root directory, Android Studio will apply those rules as well. [The EditorConfig site](#) has the details of the options that can be specified in that file.

If you reformat your code (e.g., `Ctrl-Alt-L` for Windows and Linux users), it will follow the settings you have chosen in Android Studio, including any set in `.editorconfig`.

Inlay Hints

In some of the screenshots of Kotlin code shown in this book, you may have noticed “inlay hints”:



```
private fun updateButton() {  
    val nowMs : Long ← SystemClock.elapsedRealtime()  
    val caption : String ← getString("%d seconds since started!",  
    showElapsed?.text = caption  
}
```

The image shows a Kotlin code snippet with two red arrows pointing to inlay hints. The first arrow points to the type `Long` in the declaration `val nowMs : Long`. The second arrow points to the type `String` in the declaration `val caption : String`. The text `"%d seconds since started!"` is highlighted in green.

Figure 311: Kotlin Code Snippet with Inlay Hints Highlighted

In Kotlin, we do not have to manually enter a lot of the data types that we would in Java — Kotlin’s compiler infers the types on its own. However, that does mean that we have a bunch of variables and properties for which there is no type information in our code, which makes reading that code a bit more difficult.

We can enable “inlay hints”, though. That causes Android Studio to use the compiler-inferred types, displaying them inline as if we had typed them in. The actual source code does not contain the hints, and they are not editable as text. They simply act as markers, to show you what the types are.

CONFIGURING ANDROID STUDIO

In “Editor” > “Inlay Hints” > “Kotlin”, you can decide whether to show inlay hints and for what scenarios you want them to appear:

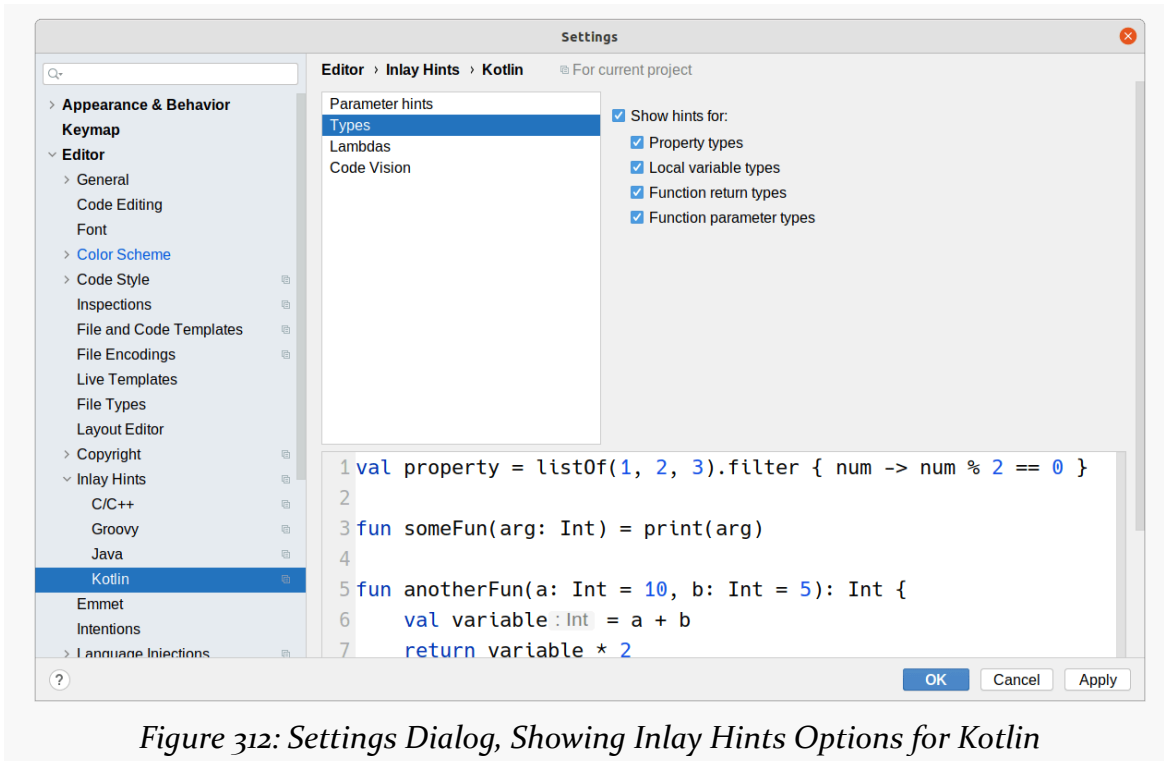


Figure 312: Settings Dialog, Showing Inlay Hints Options for Kotlin

Other Settings of Note

Another thing that some developers like to customize is the hotkey bindings for various operations (e.g., the aforementioned `Ctrl-Alt-L` to reformat the code in the current editor). There is a “Keymap” category in Settings that lets you see what all those key bindings are:

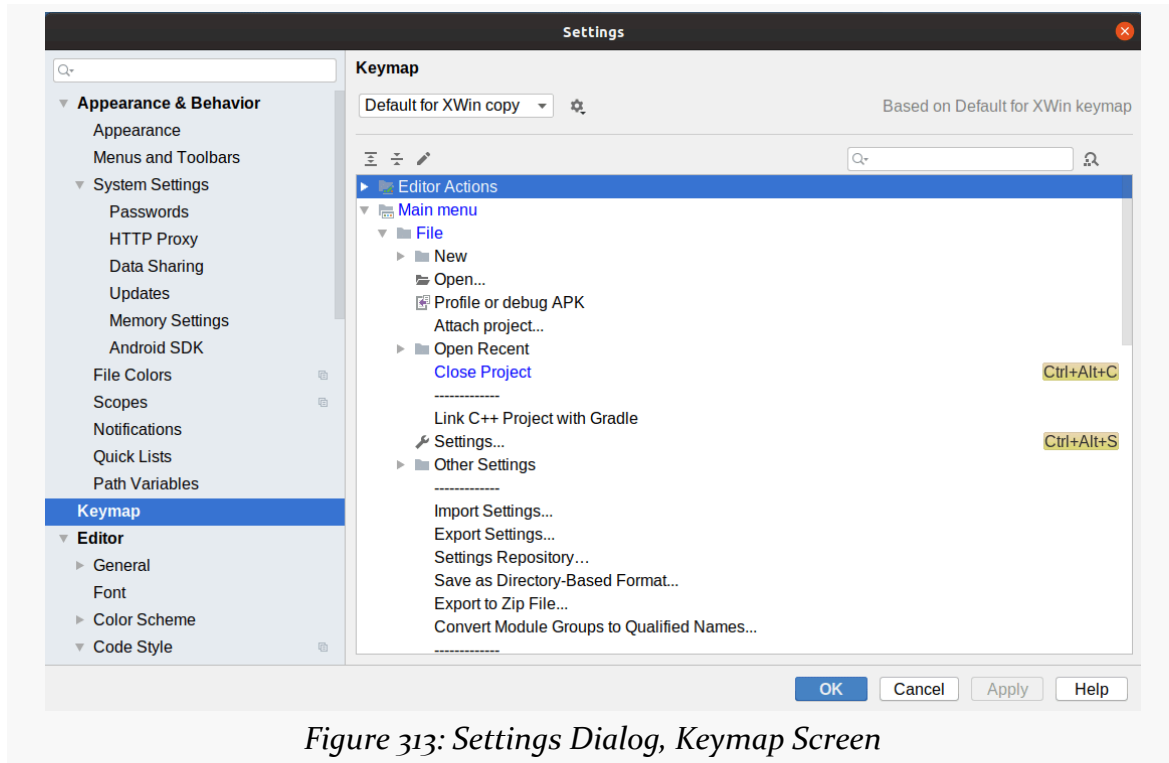


Figure 313: Settings Dialog, Keymap Screen

CONFIGURING ANDROID STUDIO

If you double-click on an action, you will be able to clear existing bindings or set up new ones:

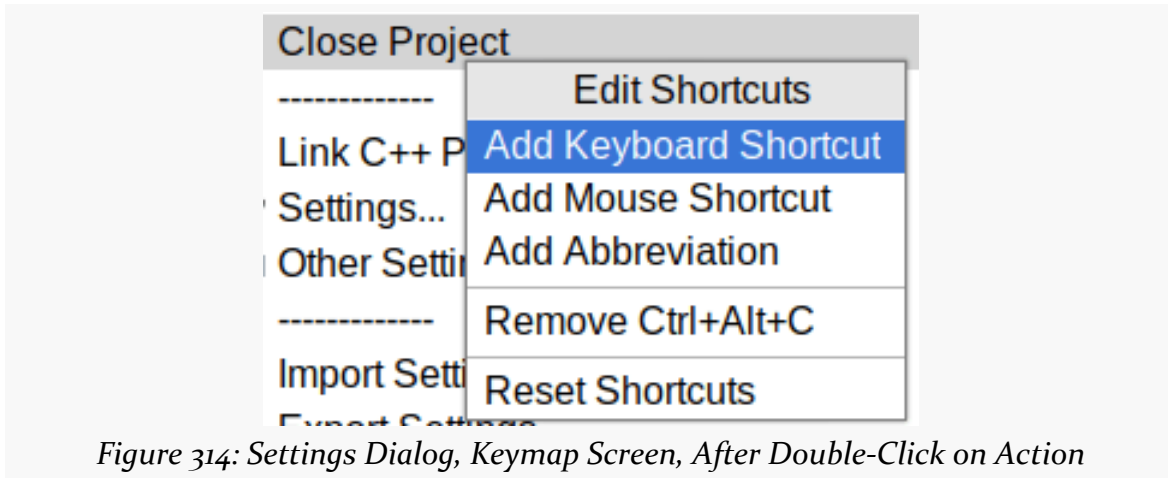


Figure 314: Settings Dialog, Keymap Screen, After Double-Click on Action

Also, Android Studio likes to use a lot of memory. If you have a large project, you may start running out of memory.

In the IDE main window, in the status bar, you can see the amount of heap space Android Studio is using, and what the overall limit is:

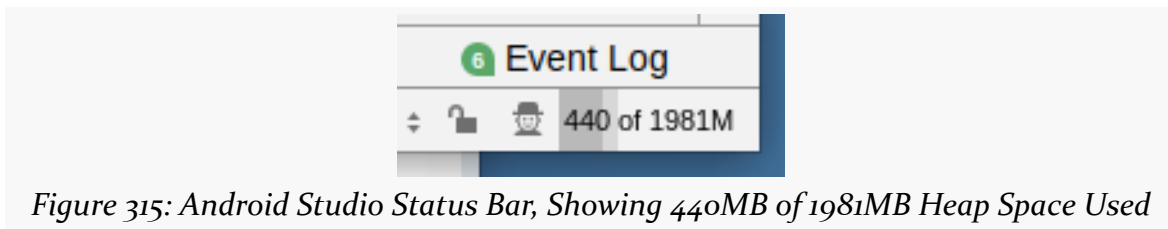


Figure 315: Android Studio Status Bar, Showing 440MB of 1981MB Heap Space Used

CONFIGURING ANDROID STUDIO

If you find that you are coming close to hitting the limit, you can adjust the amount of heap space that Android Studio is allowed to request from the OS:

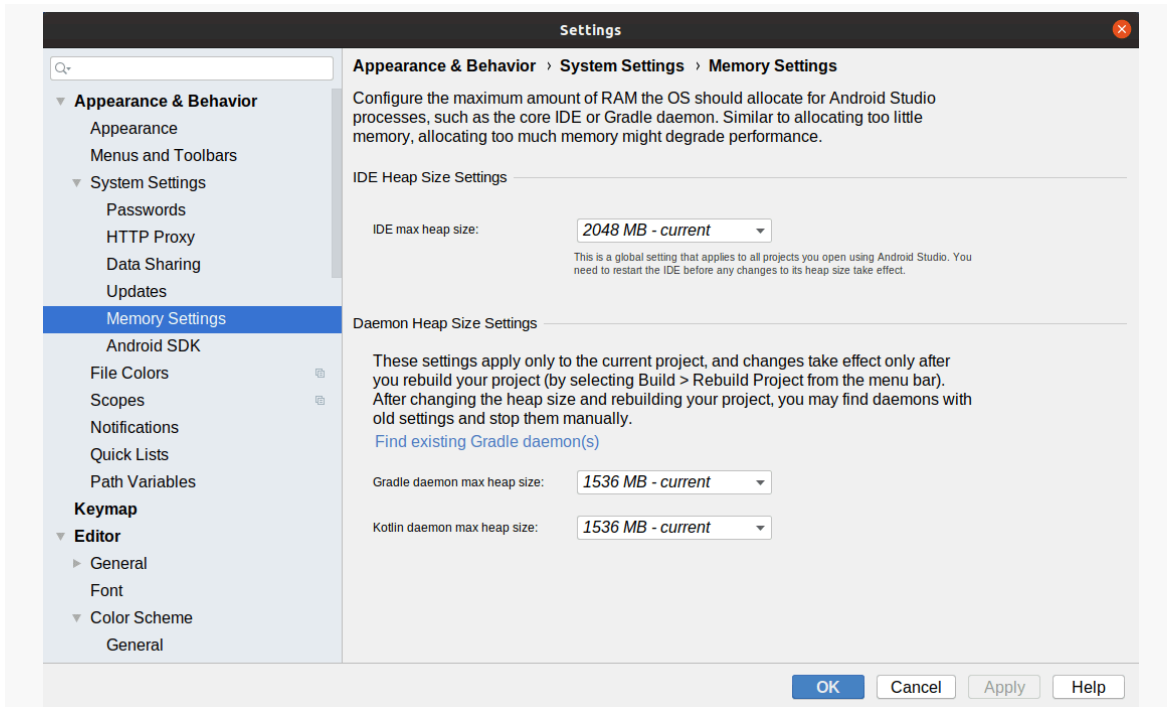


Figure 316: Settings Dialog, Memory Screen

Coping with New Android Versions

Since 2008, every year we have had a new Android version... except for those years when we had more than one new Android version.

As a result, dealing with new Android versions is a common process and a common source of headaches for Android developers. This chapter helps to explain what you should expect and how generally to go about dealing with changes.

The March of the Versions

There have been four periods of Android releases, with different release frequencies.

2008-2010: So Many Releases

Early on, we had lots of releases:

- Android 1.0
- Android 1.1
- Android 1.5
- Android 1.6
- Android 2.0
- Android 2.1
- Android 2.2
- Android 2.3

This does not include various patch releases, such as Android 2.3.3. A new Android release with new functionality was never more than a few months away.

2011: The Rise of the Tablet

2011 saw a similar cadence of releases. Three of them — 3.0, 3.1, and 3.2 — were exclusively available for tablets. Android 4.0 brought some of the Android 3.x changes (e.g., the “holo” widget theme) to phones.

2012-2013: May the Fours Be With You

The next two years saw four point releases in Android 4.x: 4.1, 4.2, 4.3, and 4.4. While that is a lot, it still represents a bit of a slowdown, with only two releases per year.

2014 Onwards: Serenity Now

Since the release of Android 5.0 in 2014, the frequency of releases has slowed to one or two per year. Each year has received a new major release, and some years also received a “.1” minor release. However, the trend has been towards a single release per year.

The Typical Release Process

Not only has the release cadence slowed down, but it has normalized in the past several years. Starting with Android 5.0, we have had a fairly consistent pattern for when and how Google releases new major versions of Android.

Step #1: The First Developer Preview

In February or March, Google will ship a developer preview or beta release of the new version. Historically, this has been given a letter designation (e.g., 2019’s developer preview was Android Q, 2020’s is Android R). The first developer preview is often referred to as “DP1” for simplicity.

A developer preview includes:

- Explanations of changes to Android, including new features and new restrictions
- An SDK version based on the name that we can use for `compileSdkVersion` (e.g., `android-R`) and `targetSdkVersion` (e.g., `'R'`)
- An emulator that runs a version of Android that supports the new SDK
- Firmware images that can be loaded onto select Android devices (mostly

some of Google's Pixel devices) that allow those devices to run the preview version of Android

Early adopters can download these things and start experimenting with the changes.

Shortly after this release, the author of this book will start publishing [blog posts](#) about the new Android version, such as [this post](#) on Android R DP1. And, shortly after that, CommonsWare will have a new book for that new Android version, akin to [2019's *Elements of Android Q*](#) and [2020's *Elements of Android R*](#).

Step #2: Additional Early Previews

Roughly once a month thereafter, Google will release another developer preview. This will contain updates to the same materials that were published as the previous developer preview. Notably, the updated versions will fix various bugs that were reported by early adopters... at least those bugs that are considered by Google to be bugs and not “working as intended”.

Step #3: Google I/O

Google holds their primary developer conference — Google I/O — in May. In recent years, the conference is held in the Shoreline Amphitheatre complex, just down the road from Google's Mountain View offices. Google also livestreams many of the sessions on YouTube and publishes recordings of them afterwards.

Several of the Google I/O sessions will be about the new Android version. Usually, there is a “What's New in Android” that covers all the major changes. And, usually, there are many dedicated sessions on individual new features or specific changes that developers need to take into account.

In 2020, Google I/O was cancelled on account of the COVID-19 pandemic.

Step #4: The Stable API Preview

A month or two after Google I/O, Google will ship a developer preview where the APIs defined in the Android SDK for this version will be considered final and should not be changing anymore.

Also, we will get official word of the next API level number, which is always the previous API level number plus one. For example, Android 10 was API Level 29, following on Android 9's API Level 28. Also, we will be able to switch to these

numbers for the `compileSdkVersion` and `targetSdkVersion` values for our projects.

Step #5: Shipping to Production

After that — where the timing has ranged from August to November — Google will ship the new Android version to the newer generations of Pixel devices. Other device manufacturers will start shipping updates to some of their devices in the coming months. A few devices beyond the Pixels may get updates rapidly, but usually manufacturers delay shipping OS updates by many months, if ever.

Sometime after this, the author of this book will start updating it for the new Android version, based on the material from the corresponding *Elements* book published earlier.

Step #6: New Official Hardware

Sometime around the production release date, Google will have a press event where they will unveil their new hardware for the year. This usually includes some new Pixel devices. These then tend to ship a few weeks after the launch event.

This is important because occasionally Android OS updates include features that are tied to specific hardware capabilities. If existing Pixels lack that hardware, these new Pixel devices might be the first ones where we can really use those new OS features.

Things to Worry About

For the past several years, each new version of Android represents an incremental change over the previous version. Simply put, there are too many Android users for Android to risk huge changes.

However, even incremental changes may require some amount of response from you as an app developer.

Breaking Changes

The type of change that worries developers the most are “breaking” changes, ones that cause your app to no longer function correctly. Occasionally, this results in crashes. More often, it results in your app just not receiving the data that it expects or otherwise getting what you planned on from the OS.

Examples of breaking changes include:

- Android 6.0's introduction of [runtime permissions](#)
- Android 7.0's ban on `Uri` values with a file scheme, necessitating the use of `FileProvider` for sharing files with other apps
- Android 9's ban on cleartext (e.g., `http`) communications, in favor of encrypted (e.g., `https`) communications
- Android 10's restrictions on [external storage](#)

Immediate Breakage

Sometimes, these changes will affect your app immediately. This is relatively uncommon and usually is not in the APIs that we have covered in this book.

If you find out about such a breaking change, you need to address this before Google ships the new version of Android to its Pixel devices and other manufacturers start updating their devices to match. Even sooner than that you will start running into problems, as some “power users” will be playing with the developer preview builds, even if they are not actually developers.

Eventual Breakage

The more common type of breakage is tied to your `targetSdkVersion`. When your `targetSdkVersion` reaches the API level for the new release, then you are subject to the changes. Apps with an older `targetSdkVersion` would not be subject to the changes.

While this approach has been used for many years, the results have changed recently. That is because Google now enforces a minimum `targetSdkVersion` for apps on the Play Store, and other app distributors are starting to do the same. Approximately one year after an Android version ships to devices, you will need to have a `targetSdkVersion` matching (or exceeding) that Android version's API level. Basically, Google prevents you from shipping an update to your app, or shipping a new app, unless you meet the `targetSdkVersion` requirement.

The good news is that gives you nearly 1.5 years from the time of the first developer preview before you have to address changes that are tied to `targetSdkVersion`. The bad news is that developers might well forget what changes are needed by then. Where possible, try to address even these delayed changes as soon as is practical, rather than wait until the last minute.

Stuff Users Will Expect

Sometimes, Google does not force you to change your app, but your users might. That is because sometimes your app may need to be adjusted to allow certain new Android capabilities to work, and users might complain if your app fails to do so.

A good example is Android 10's dark mode. As we saw back in [the chapter on styles and themes](#), Android 10 offers a “system wide” dark mode that users can enable. In reality, Android 10 only makes system UI dark, plus it tells visible activities about the change. It is up to app developers to actually *make* a change to support dark mode. Apps that do not do so simply do not change when dark mode is toggled on or off. Some users will complain about that missing capability, and if they complain in highly-visible places (e.g., Play Store reviews), that may have an impact on your app's success.

Usually, these sorts of user-facing visible features get touted a lot in blog posts and other coverage of the new Android version. If you are paying attention to the new release, you will have a fairly good sense of what users will expect. How easy it will be for your app to adopt that new feature may vary, but you should budget time for it as soon as is practical.

Deciding Where to Go From Here

Congratulations on completing the book!

(or, at the very least, congratulations for turning to this chapter!)

At this point, you should be “up to speed” on the basics of developing Android apps. However, Android is vast, and this book only scratches the surface. For example, while you may think this book is large, [The Busy Coder’s Guide to Android Development](#) was about five times larger... and *it* did not cover everything in Android.

So, here are ways to learn more about other areas of Android and to get help in your ongoing development efforts.

The Rest of the Books

If you subscribed to [the Warescription](#), there are many more books that you can read as needed. These include:

- [Exploring Android](#), the hands-on counterpart to this book, walking you through building an app step-by-step
- [Elements of Kotlin](#) and [Elements of Kotlin Coroutines](#), if you need more information about the up-and-coming options for writing apps
- [Elements of Android Room](#), for more breadth and depth on Google’s object/relational mapping engine for SQLite

The “Elements” series also covers books about newer Android versions, written as they come out, such as 2020’s [Elements of Android R](#).

The aforementioned [The Busy Coder's Guide to Android Development](#) and other older books are also available to you. Mostly, though, *The Busy Coder's Guide* is for specialized topic areas in Android app development.

Note that all of the books are searchable via [the Warescription site](#), so you can search on class names or subjects of interest and see what is available across the entire CommonsWare library.

You might want to follow [the CommonsBlog](#), both for announcements of new books and updates to books, along with lots of additional information on Android app development.

Android Developer Support

You've got questions. That's understandable!

There are many places where you can get your Android app development questions answered, beyond just searching on your topic in a search engine:

- [Stack Overflow's android tag](#) has over a million questions, many with answers. Questions that you ask there have a decent chance of getting an answer, particularly if the question is well-written. Warescription subscribers can also “ping” the author of this book, [asking for help with a specific Stack Overflow question](#).
- Warescription subscribers can also [ask for posting rights](#) to [the Android Development category in the CommonsWare Community](#). There, you can ask all sorts of Android app development questions, and the author of this book will try to answer them.
- Warescription subscribers also have access to [“office hours” live chats](#) — you can see transcripts of past chats [here](#)
- If you have questions about Kotlin, JetBrains operates [a Slack workspace](#) — you can ask for an invitation [here](#)
- There are many Android developer discussion boards in [a variety of languages](#) to choose from as well

Major Conferences

As was noted in [the previous chapter](#), Google's main developer conference is Google I/O. For the past several years, it has been held in May. In 2018 and 2019, Google has also organized the Android Developer Summit, a two-day Android-focused

DECIDING WHERE TO GO FROM HERE

conference held in October or November. In both cases, the conference sessions are published on YouTube after the event.

The largest collection of independent conferences is [the droidcon series](#). Past events have been held in Europe, Asia, Africa, and North America. Each event is independently operated, and where and how they publish conference videos varies by event.

There are many other Android-focused events outside of droidcon, such as:

- [360|AnDev](#) in Denver
- [Android Makers](#) in Paris
- The [Android Summit](#) (not to be confused with Google's Android Developer Summit) in Washington DC
- [Chicago Roboto](#), in Chicago

You can find a list of upcoming Android conferences [here](#). Note that many of these events have moved online or are canceled for 2020, due to the ongoing COVID-19 pandemic.

