4-е издание
Обновлено для Java 8

Патрик Нимейер
Дэниэл Леук



Программирование на JAVA

Исчерпывающее руководство для профессионалов



- **Все о библиотеках классов, приемах программирования и идиомах**
- **Новые мощные функциональные интерфейсы**
- **Руководство для программистов с множеством реальных примеров**

O'REILLY®

ЭКСМО

УДК 004.42(075.4)
ББК 32.973-018.2
Н 67

Patrick Niemeyer, Daniel Leuck
LEARNING JAVA

Authorized Russian translation of the English edition of Learning Java,
4th Edition (ISBN 9781449319243) © 2013 Patrick Niemeyer, Daniel Leuck.
This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

Нимейер, Патрик.

Н 67 Программирование на Java / Патрик Нимейер, Дэниэл Леук ; [пер. с англ. М. А. Райтмана]. – Москва : Эксмо, 2014. – 1216 с. – (Мировой компьютерный бестселлер).

Комплексное руководство по освоению языка программирования Java с многочисленными подробными и реалистичными примерами. Если вы – начинающий программист, то книга станет незаменимым помощником для того, чтобы стать профессионалом, а если у вас уже есть опыт работы – вы узнаете о многочисленных тонкостях и разберетесь в самых новых средствах для создания приложений и сервисов. Описаны новинки, появившиеся в Java 8.

УДК 004.42(075.4)
ББК 32.973-018.2

Производственно-практическое издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Патрик Нимейер, Дэниэл Леук

ПРОГРАММИРОВАНИЕ НА JAVA
(орыс тілінде)

Директор редакции *Е. Капьёв*
Ответственный редактор *В. Обручев*
Художественный редактор *Г. Федотов*

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86, 8 (495) 956-39-21.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел. 8 (495) 411-68-86, 8 (495) 956-39-21
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Қазақстан Республикасында дистрибьютор және өнім бойынша
арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 2 51 59 89,90,91,92, факс: 8 (727) 251 58 12 вн. 107; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить по адресу: http://eksmo.ru/certification/
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 12.08.2014. Формат 70x100$^{1}/_{16}$. Печать офсетная.
Усл. печ. л. 98,52. Тираж 2000 экз. Заказ

ISBN 978-5-699-71504-6

ISBN 978-5-699-71504-6

© Райтман М.А., перевод на русский язык, 2014
© Оформление. ООО «Издательство «Эксмо», 2014

ОГЛАВЛЕНИЕ

Предисловие 17
Кому следует читать эту книгу 18
Новые разработки 18
Новое в этом издании 19
Использование этой книги 20
Веб-ресурсы 22
Примеры к книге 22
Условные обозначения, использованные в этой книге 22
Использование примеров кода 23
Благодарности 24

Глава 1. Современный язык 25
Появление Java 26
Происхождение Java 26
Становление 28
Виртуальная машина 29
Java в сравнении с другими языками 33
Безопасное проектирование 37
Упрощение, упрощение, упрощение 38
Безопасность типов и связывание методов 39
Инкрементная разработка 41
Управление динамической памятью 41
Обработка ошибок 43
Потоки 43
Масштабируемость 44
Безопасность реализации 45
Верификатор 47
Загрузчики классов 48
Диспетчеры безопасности 49
Приложение и безопасность на уровне пользователя 50
История и перспективы развития Java 51
Прошлое: Java 1.0 — Java 1.6 52
Java 7 53
Настоящее: Java 8 56
Будущее 74
Доступность 74

Глава 2. Первое применение 75
Инструменты и среда Java 75
Конфигурация Eclipse и создание проекта 77
Импорт примеров по изучению Java 78
`HelloJava` 79
Классы 82
Метод **`main()`** 83
Классы и объекты 85
Переменные и типы классов 86
`HelloComponent` 87
Наследование 88
Класс **`JComponent`** 89
Связи и поиск виновных 90
Пакеты и инструкции импорта 91
Метод **`paintComponent()`** 93
`HelloJava2`: продолжение 95
Переменные экземпляра 96
Конструкторы 97
События 99
Метод **`repaint()`** 102
Интерфейсы 103

HelloJava3: атака класса **JButton**! 104
Перегрузка метода 107
Компоненты 107
Контейнеры 108
Диспетчеры компоновки 109
Подклассы и подтипы 110
Больше событий и интерфейсов 110
Замечание относительно цвета 112
Статические члены 112
Массивы 113
Методы для класса **Color** 114
HelloJava4: месть Netscape 116
Потоки выполнения 118
Класс **Thread** 119
Интерфейс **Runnable** 120
Запуск потока 120
Выполнение кода в классе **Thread** 121
Исключения 122
Синхронизация 124
Глава 3. Инструменты для работы 126
Среда JDK 126
Виртуальная машина Java 127
Запуск приложения Java 128
Свойства системы 129
Переменная среды **Classpath** 130
Утилита **javap** 132
Компилятор Java 133
Файлы JAR 135
Сжатие файлов 135
Утилита **jar** 136
Утилита **pack200** 140
Файлы политики 141
Стандартный диспетчер безопасности 141
Утилита **policytool** 142
Использование файла политики со стандартным диспетчером безопасности 144
Глава 4. Язык Java 145
Кодировка текста 145
Комментарии 147
Комментарии Javadoc 147
Типы 149
Простые типы 150
Ссылочные типы 155
Слово о строках 158
Операторы и выражения 158
Операторы 159
Выражения 168
Исключения 173
Исключения и классы ошибок 174
Обработка исключений 177
Восхождение исключений 179
Трассировка стека 180
Проверяемые и непроверяемые исключения 181
Выброс исключений 183
Разумное использование оператора try 187
Оператор **finally** 188
Оператор **try** с очисткой ресурсов 189
Проблемы, связанные с производительностью 191
Утверждения 192
Включение и выключение утверждений 193
Использование утверждений 194

Массивы . . . 196
Типы массивов . . . 197
Создание и инициализация массива . . . 198
Использование массивов . . . 200
Анонимные массивы . . . 202
Многомерные массивы . . . 203
Внутри массивов . . . 205
Глава 5. Объекты в Java . . . 206
Классы . . . 207
Доступ к полям и методам . . . 209
Статические члены . . . 211
Методы . . . 215
Локальные переменные . . . 216
Сокрытие . . . 216
Статические методы . . . 218
Инициализация локальных переменных . . . 219
Передача аргументов и ссылки . . . 220
Обертки для простых типов . . . 222
Автоматическое упаковывание и распаковывание простых типов . . . 224
Список аргументов переменной длины . . . 226
Перегрузка метода . . . 227
Создание объектов . . . 229
Конструкторы . . . 230
Работа с перегруженными конструкторами . . . 231
Статические и нестатические блоки инициализаторов . . . 233
Разрушение объектов . . . 235
Сборка мусора . . . 235
Завершение . . . 236
Слабые и мягкие ссылки . . . 237
Перечисления . . . 239
Значения класса `Enum` . . . 240
Изменение перечислений . . . 241
Глава 6. Отношения между классами . . . 243
Создание подклассов и наследование . . . 243
Перекрытие переменных . . . 246
Переопределение методов . . . 248
Специальные ссылки: `this` и `super` . . . 256
Приведение типов . . . 257
Использование конструкторов надклассов . . . 260
Полное разоблачение: конструкторы и инициализация . . . 261
Абстрактные методы и классы . . . 262
Интерфейсы . . . 263
Интерфейсы и обратные вызовы . . . 266
Переменные интерфейса . . . 267
Субинтерфейсы . . . 269
Пакеты и блоки компиляции . . . 270
Блоки компиляции . . . 270
Имена пакетов . . . 271
Видимость класса . . . 272
Импорт классов . . . 272
Видимость переменных и методов . . . 275
Базовые модификаторы доступа . . . 275
Подклассы и видимость . . . 277
Интерфейсы и видимость . . . 278
Массивы и иерархия классов . . . 278
Исключение `ArrayStoreException` . . . 280
Внутренние классы . . . 280
Внутренние классы и адаптеры . . . 283
Внутренние классы внутри методов . . . 285

Глава 7. Работа с объектами и классами 294
Класс `Object` 294
Равенство и эквивалентность 295
Хеш-коды 296
Клонирование объектов 297
Класс Class 301
Отражение 304
Модификаторы и безопасность 307
Доступ к полям 308
Доступ к методам 310
Доступ к конструкторам 312
Что насчет массивов? 313
Доступ к информации обобщенного типа 313
Доступ к данным аннотаций 314
Адаптеры с динамическим интерфейсом 315
Для чего подходит отражение? 317
Аннотации 318
Использование аннотаций 319
Стандартные аннотации 321
Инструмент apt 321
Глава 8. Обобщения 323
Контейнеры: новый уровень абстракций 324
Можно ли исправить контейнеры? 325
Ввод обобщений 326
Разговор о типах 330
«Ложки не существует» 330
Стирание 331
Необработанные типы 333
Отношения параметризированных типов 335
Почему `List<Date>` не является `List<Object>`? 337
Приведение типов 338
Написание обобщенных классов 340
Переменная типа 340
Наследование обобщений 341
Исключения и обобщения 342
Ограничения, связанные с типом параметра 343
Границы 346
Стирание и границы (работа со старым кодом) 347
Групповые символы 348
Супертип для всех реализаций 349
Ограниченные групповые типы 349
Групповые типы в контексте контейнеров 350
Нижние границы 351
Чтение, запись и арифметические операции 352
`<?>`, `<Object>` и необработанный тип 354
Отношения группового типа 355
Обобщенные методы 357
Знакомство с обобщенными методами 358
Выведение типа из аргументов 359
Выведение типа из контекста назначений 360
Явный вызов типа 361
Перехват группового типа 361
Выбор между групповыми типами и обобщенными методами 362
Массивы параметризованных типов 363
Использование типов массивов 364
Чем хороши массивы обобщенных типов? 365
Групповые типы в типах массивов 365
Пример: класс `Enum` 366
Пример: метод `sort()` 368
Заключение 369

Глава 9. Потоки выполнения 370
Введение в потоки 371
Класс **Thread** Class и интерфейс **Runnable** 372
Управление потоками 377
Смерть потока 380
Выполнение апплета в нескольких потоках 382
Отслеживание проблем. 384
Синхронизация. 385
Синхронизированный доступ к методам 386
Класс доступа и переменные экземпляров объекта из потоков многопоточных программ 389
Методы **wait()** и **notify()** 390
Передача сообщений 393
Объекты **ThreadLocal** 398
Планирование и приоритеты 399
Состояние потока 401
Временные интервалы 402
Приоритеты 403
Возвращение 404
Группы потоков 405
Работа с классом **ThreadGroup** 406
Непроверяемые исключения 406
Выполнение потока 407
Издержки синхронизации 407
Потребление ресурсов потоком. 408
Средства для поддержки параллелизма 409
Исполнители 411
Блокировки 427
Средства синхронизации. 430
Атомарные операции 436
Заключение 439

Глава 10. Работа с текстом 440
API, связанные с текстом 441
Строки 441
Создание строк 442
Создание строк на основе различных типов 444
Сравнение строк 445
Поиск 447
Редактирование 448
Краткое описание методов объекта **String** 449
Объекты **StringBuilder** и **StringBuffer** 450
Интернационализация 452
Класс **java.util.Locale** 452
Наборы ресурсов 453
Разбор и форматирование текста 456
Разбор простых чисел. 457
Разметка текста. 459
Форматирование **Printf-Style** 461
Программа форматирования 462
Строка форматирования 462
Преобразования строк 464
Простые и числовые преобразования 466
Флаги 468
Прочие положения. 469
Форматирование при помощи пакета **java.text** 469
Средство **MessageFormat**. 472
Регулярные выражения 474
Язык регулярных выражений. 475
java.util.regex API 488

Глава 11. Основные инструменты 496
Математические инструменты 496
Класс `java.lang.Math` 497
Большие/точные числа 499
Компоненты числа с плавающей запятой 500
Случайные числа 501
Дата и время 502
Работа с календарями 503
Часовые пояса 505
Разбор и форматирование даты при помощи класса `DateFormat` 508
Форматирование даты и времени в стиле метода `printf` 510
Таймеры 512
Коллекции 514
Интерфейс `Collection` 515
Итератор 519
Виды коллекций 520
Интерфейс `Map` 524
Реализации коллекций 527
Хеш-коды и ключи 533
Синхронизированные и несинхронизированные коллекции 535
Коллекции, предназначенные только/преимущественно для чтения 538
Класс `WeakHashMap` 538
Классы `EnumSet` и `EnumMap` 539
Сортировка коллекций 539
Захватывающий пример 540
Класс Properties 542
Загрузка и сохранение 543
Системные свойства 545
Технология Preferences API 546
Применение технологии Preferences API для классов 547
Хранилище конфигурационных данных 548
Уведомления об изменениях 549
API-интерфейсы для ведения журнала сообщений 550
Обзор 551
Уровни журнальных сообщений 554
Простой пример 555
Изменение свойств журнала 556
Класс `Logger` 558
Производительность 559
Интерфейс Observer и класс Observable 560

Глава 12. Средства ввода/вывода 563
Потоки ввода/вывода 563
Основы ввода/вывода 567
Потоки символов 570
Обертки для потоков 572
Конвейеры 577
Чтение и запись потоков строк 579
Реализация фильтрующего потока 581
Файловый ввод/вывод 583
Класс `java.io.File` 583
Файловые потоки 590
Класс `RandomAccessFile` 594
Пути к ресурсам 595
API-интерфейс пакета NIO для работы с файлами 598
Классы `FileSystem` и `Path` 599
Файловые операции, предоставляемые пакетом NIO 601
Операции с директориями 605
Отслеживание путей 608
Сериализация 609
Инициализация объектов с помощью метода `readObject()` 611
Идентификатор `serialVersionUID` 613

Сжатие данных . . . 614
Архивы и сжатие данных . . . 614
Распаковка данных . . . 617
ZIP-архив в качестве файловой системы . . . 619
Пакет NIO . . . 621
Асинхронный ввод/вывод . . . 621
Производительность . . . 622
Блокирование файлов и отображение их в память . . . 622
Каналы . . . 622
Буферы . . . 623
Кодирование и декодирование символов . . . 628
Класс `FileChannel` . . . 631
Масштабируемый ввод/вывод и пакет NIO . . . 639
Глава 13. Сетевое программирование . . . 641
Сокеты . . . 643
Клиенты и серверы . . . 645
Клиентское приложение `DateAtHost` . . . 651
Серверное приложение TinyHttpd . . . 653
Свойства сокетов . . . 659
Прокси и брандмауэры . . . 661
Датаграммные сокеты . . . 664
Апплет HeartBeat . . . 665
Класс `InetAddress` . . . 670
Простой протокол для передачи сериализованных объектов . . . 671
Простой сервер для обмена объектами . . . 672
Удаленный вызов методов . . . 677
Реальное применение . . . 678
Удаленные и локальные объекты . . . 678
Пример использования технологии RMI . . . 683
RMI и CORBA . . . 693
Масштабируемый ввод/вывод на основе пакета NIO . . . 693
Выборочные каналы . . . 695
Использование селекторов . . . 696
Приложение LargerHttpd . . . 698
Неблокирующие операции на клиентской стороне . . . 704
Глава 14. Веб-программирование . . . 705
Единый указатель ресурсов (URL) . . . 705
Класс `URL` . . . 706
Потоковая передача данных . . . 708
Получение данных в виде объекта . . . 709
Управление соединениями . . . 710
Использование обработчиков в реальных задачах . . . 711
Технологии обработки данных, на которые стоит обратить внимание . . . 712
Взаимодействие с веб-приложениями . . . 713
Использование запросов типа GET . . . 714
Использование запросов типа POST . . . 715
Класс `HttpURLConnection` . . . 719
Технология SSL и безопасность при взаимодействии с веб-приложениями . . . 719
Идентификаторы URL, URN и URI . . . 720
Веб-сервисы . . . 721
Технология XML-RPC . . . 722
Язык WSDL . . . 722
Инструменты для работы с веб-сервисами . . . 723
Клиент для метеослужбы . . . 723
Глава 15. Веб-приложения и веб-сервисы . . . 726
Технологии, лежащие в основе веб-приложений . . . 727
Многостраничные и одностраничные приложения . . . 728
JSP-сценарии . . . 729
XML и XSL . . . 730
Фреймворки для создания веб-приложений . . . 730

Google Web Toolkit . . . 731
HTML5, AJAX и т. д. . . . 731
Веб-приложения на платформе Java . . . 732
Жизненный цикл сервлета . . . 733
Сервлеты . . . 734
Сервлет **HelloClient** . . . 735
Ответ сервлета . . . 738
Параметры сервлета . . . 739
Сервлет **ShowParameters** . . . 741
Управление пользовательскими сессиями . . . 743
Сервлет **ShowSession** . . . 744
Сервлет **ShoppingCart** . . . 747
Файлы cookie . . . 750
Интерфейс **ServletContext** . . . 752
Асинхронные сервлеты . . . 753
WAR-файлы и развертывание веб-приложений . . . 757
Конфигурирование с помощью файла web.xml и аннотаций . . . 759
Привязка к URL-шаблонам . . . 762
Развертывание сервлета **HelloClient** . . . 763
Вывод ошибок и индексные страницы . . . 765
Безопасность и аутентификация . . . 767
Защита ресурсов с помощью ролей безопасности . . . 768
Защищенная передача данных . . . 769
Аутентификация пользователей . . . 771
Авторизация в императивном стиле . . . 773
Фильтры сервлетов . . . 774
Простой фильтр . . . 775
Тестовый сервлет . . . 777
Объявление и привязка фильтров . . . 778
Фильтрация запроса к сервлету . . . 779
Фильтрация ответа, который возвращается сервлетом . . . 781
Создание WAR-файлов с помощью утилиты Ant . . . 785
Иерархия директорий, ориентированная на разработку . . . 786
Развертывание WAR-архивов с помощью утилиты Ant . . . 788
Реализация веб-сервисов . . . 788
Объявление сервиса . . . 789
Наш веб-сервис Echo . . . 790
Использование сервиса . . . 791
Типы данных . . . 793
Итог . . . 795
Глава 16. Библиотека Swing . . . 796
Компоненты . . . 800
Системные компоненты и внешний вид . . . 802
Архитектура MVC в пользовательских интерфейсах . . . 804
Рисование . . . 805
Включение и отключение компонентов . . . 807
Фокусируемся . . . 807
Другие методы класса **JComponent** . . . 808
Диспетчеры компоновки . . . 810
Вставки . . . 811
Порядок наложения компонентов . . . 811
Методы **revalidate()** и **doLayout()** . . . 812
Управление компонентами . . . 813
Прослушивание компонентов . . . 813
JFrame и JWindow . . . 813
Другие методы для управления окнами . . . 816
Панели . . . 816
Интеграция с настольной средой . . . 817
События . . . 820
Получатели событий и интерфейс **EventListener** . . . 821
Источники событий . . . 823
Доставка событий . . . 825
Типы событий . . . 826

Класс **java.awt.event.InputEvent** 827
Модификаторы, связанные с мышью и клавиатурой 827
События, связанные с выделением компонентов 829
Список часто используемых событий 831
Классы адаптеров 833
Шаблонные адаптеры 836
Настоящий робот в библиотеке AWT! 838
Многопоточность в библиотеке Swing 839
Глава 17. Использование компонентов библиотеки Swing 843
Кнопки и метки 843
Разметка текста в кнопках и метках с помощью формата HTML 846
Флажки и переключатели 847
Обычные и раскрывающиеся списки 851
Поле ввода со счетчиком 854
Обрамление 857
Меню 861
Контекстные меню 865
Управление контекстным меню на уровне отдельных компонентов 868
Класс **JScrollPane** 870
Класс JSplitPane 872
Класс **JTabbedPane** 873
Полосы прокрутки и ползунковые регуляторы 878
Диалоговые окна 880
Диалоговое окно выбора файлов 884
Диалоговое окно выбора цвета 887
Глава 18. Другие компоненты библиотеки Swing 888
Компоненты для работы с текстом 888
Приложение TextEntryBox 889
Форматированный текст 892
Фильтрация ввода 893
Проверка данных 895
Сезам, откройся! 897
Использование общей модели данных 898
Встроенная поддержка форматов HTML и RTF 900
Работа с текстом вручную 904
Навигация посредством выделения компонентов 906
Древовидные списки 908
Узлы и модели 909
Стандартное поведение древовидного списка 910
События в древовидном списке 910
Готовый пример 911
Таблицы 914
Раунд первый: сразу в бой 914
Раунд второй: создание модели данных 917
Раунд третий: простая электронная таблица 920
Сортировка и фильтрация 924
Печать таблиц 927
Рабочие столы 928
Изменение внешнего вида 929
Создание нестандартных компонентов 933
Генерирование событий 933
Круговой регулятор 934
Отделение модели данных от представления 938
Глава 19. Диспетчеры компоновки 940
FlowLayout 942
GridLayout 944
BorderLayout 945
BoxLayout 948
CardLayout 950
GridBagLayout 952
Класс **GridBagConstraints** 953
Координаты сетки 955

Параметр **fill** . . . 957
Заполнение нескольких строк или столбцов . . . 958
Параметры **weightx** и **weighty** . . . 960
Параметр **anchor** . . . 963
Внутренние и внешние отступы . . . 963
Относительное позиционирование . . . 965
Совместное использование разных диспетчеров компоновки . . . 966
Другие диспетчеры компоновки . . . 971
Абсолютное позиционирование . . . 972
Глава 20. Рисование с использованием 2D API . . . 973
Общая картина . . . 973
Конвейер визуализации . . . 976
Краткий тур по Java 2D . . . 979
Заполнение форм . . . 979
Рисование контура формы . . . 979
Удобные методы . . . 980
Рисование текста . . . 981
Рисование изображений . . . 981
Пример целиком . . . 983
Закрашивание форм . . . 986
Сплошные цвета . . . 986
Цветовой градиент . . . 987
Текстуры . . . 987
Системная палитра . . . 988
Штрихование контуров формы . . . 989
Использование шрифтов . . . 989
Метрики шрифта . . . 991
Вывод изображений . . . 996
Класс **Image** . . . 997
Обозреватели изображений . . . 998
Размер и масштабирование . . . 1001
Техники рисования . . . 1002
Двойная буферизация . . . 1005
Ограниченное рисование с отсечением . . . 1006
Внеэкранное рисование . . . 1009
Печать . . . 1012
Глава 21. Работа с изображениями и другими медиаданными . . . 1015
Загрузка изображений . . . 1016
Интерфейс **ImageObserver** . . . 1016
Класс **MediaTracker** . . . 1019
Класс **ImageIcon** . . . 1022
Класс **ImageIO** . . . 1023
Создание растровых данных . . . 1024
Создание анимации . . . 1025
Анатомия класса **BufferedImage** . . . 1028
Цветовые модели . . . 1031
Создание изображения . . . 1032
Обновление **BufferedImage** . . . 1034
Настройка обрабатываемых изображений . . . 1038
Как работает приложение **ImageProcessor** . . . 1041
Преобразование объекта **Image** в объект **BufferedImage** . . . 1042
Использование класса **RescaleOp** . . . 1042
Использование класса **AffineTransformOp** . . . 1043
Сохранение обрабатываемого изображения . . . 1044
Работа со звуком . . . 1045
Java Media Framework . . . 1047
Глава 22. JavaBeans . . . 1050
Что такое bean-компонент? . . . 1050
Состав bean-компонента . . . 1052
NetBeans IDE . . . 1053
Установка и запуск NetBeans . . . 1054

Свойства и настройки. 1057
Подключения и адаптеры событий 1060
Добавление таймера в приложение **Juggler** 1061
Движение молекул. 1063
Связывание свойств. 1065
Принудительные свойства 1067
Создание bean-компонентов 1067
Bean-компонент **Dial** 1068
Разработка шаблонов для свойств 1071
Ограничения визуальной разработки. 1072
Создание сериализованой версии кода 1073
Использование класса **BeanInfo** 1074
Получение информации о свойствах 1075
Получение информации о событиях. 1076
Добавление значков 1077
Создание настройщиков и редакторов свойств 1078
Программирование вручную с применением bean-компонентов 1079
Установка bean-компонентов и управление типами 1079
Работа с сериализованными bean-компонентами 1080
Отслеживание событий выполнения с применением рефлексии. 1083
BeanContext и **BeanContextServices** 1085
Java Activation Framework 1086
Технология JavaBeans и фреймворки промышленного уровня, основанные на обычных объектах 1086

Глава 23. Апплеты 1088
Политика браузерных приложений 1089
Поддержка апплетов и Java-плагины 1090
Класс **JApplet** 1091
Жизненный цикл апплета 1092
«Песочница» безопасности для апплетов 1096
Обзор ресурсов апплетов. 1098
Элемент **applet** 1103
Атрибуты. 1103
Параметры 1104
¿Habla Applet? 1105
Полная версия тега **<applet>** 1105
Загрузка файлов классов 1107
Пакеты 1108
appletviewer 1109
Java Web Start. 1109
Заключение 1110

Глава 24. XML 1111
Обзор 1111
Немного предыстории 1112
Сравнение текстовых и двоичных данных 1113
Универсальный синтаксический анализатор 1114
Положение языка XML. 1114
XML API 1115
XML и веб-браузеры 1115
Основы XML 1115
Атрибуты. 1116
XML-документы 1117
Кодировка 1117
Пространство имен 1118
Проверка 1120
HTML в XHTML. 1120
SAX 1121
SAX API 1121
Создание модели с использованием SAX 1123
XMLEncoder/Decoder 1131

DOM 1132
DOM API 1132
Тест-драйв DOM 1134
Создание XML с DOM 1135
JDOM 1136
XPath 1138
Узлы 1138
Предикаты 1139
Функции 1140
XPath API 1141
XMLGrep 1142
XInclude 1144
Подключение XInclude 1145
Проверка документов 1146
Использование средств проверки документов 1146
Элементы языка DTD 1147
XML Schema 1149
Верификация API 1154
Связывание и генерация кода JAXB 1157
Описание нашей модели 1157
Создание модели Java из XML Schema 1164
Создание XML-схемы из Java-модели 1164
Преобразование документов с использованием XSL/XSLT 1165
Основы XSL 1166
Преобразование каталога 1168
Преобразование XLS 1170
XLS в браузере 1172
Веб-сервисы 1172
Пара слов в заключение 1173

Приложение А. Среда разработки Eclipse 1174
Войны IDE 1175
Знакомство с Eclipse 1175
Импорт учебных примеров Java 1176
Использование Eclipse 1177
Создание исходников 1178
Интерфейс 1179
Запуск примеров 1180
Сборка примеров в Ant-Based 1181
Отдельные примеры 1182
Функции Eclipse 1182
Создание ярлыков 1182
Автоматическая коррекция 1183
Реорганизация кода 1184
Сравнение файлов 1185
Организация импорта 1185
Форматирование исходного кода 1185
Заключение 1186

Приложение Б. BeanShell: Написание сценариев Java 1187
Запуск BeanShell 1187
Операторы и выражения Java 1188
Импорт 1189
Команды BeanShell 1190
Сценарии методов и объектов 1191
Сценарии интерфейсов и адаптеров 1192
Изменение директории classpath 1193
Дополнительные сведения 1194

Глоссарий 1195

Об авторах 1210

Предметный указатель 1211

ПРЕДИСЛОВИЕ

Перед вами книга о языке и среде программирования Java. Являетесь ли вы разработчиком программного обеспечения или просто пользуетесь Всемирной паутиной в повседневной жизни, вы, несомненно, слышали о языке Java. Его появление было одной из самых волнующих разработок в истории Всемирной паутины, а приложения языка Java сильно ускорили рост бизнеса во Всемирной паутине за последние 15 лет. Java, пожалуй, самый популярный язык программирования в мире, и его используют миллионы разработчиков на всех мыслимых типах компьютеров. За последнее десятилетие язык Java обогнал такие языки, как C++ и Visual Basic, с точки зрения требований к разработке и фактически стал основным языком для определенных типов разработки, в особенности для сервисов, использующих интернет-технологии. Многие университеты теперь обучают Java на своих начальных курсах вместе с другими важными современными языками. Возможно, в таком университете учитесь и вы...

Эта книга преподаст вам основы Java, такие как интерфейсы API. Книга старается соответствовать своему названию, включая описание языка Java и его библиотек классов, техник программирования и идиом. Мы копнем глубоко в интересных областях и по крайней мере поверхностно осветим остальной материал.

По возможности мы предлагаем захватывающие, реалистичные и забавные примеры и избегаем простой каталогизации свойств. Примеры незамысловатые, но намекают на то, что еще можно сделать. Мы не будем разрабатывать очередное великое «убийственное приложение» на этих страницах, но надеемся дать вам отправную точку для многих часов экспериментирования и вдохновенной возни, которые приведут вас к созданию такого приложения своими силами.

Материал, защищенный авторским правом

Кому следует читать эту книгу

Эта книга создана для специалистов в компьютерной науке, студентов, людей с техническим образованием и финских хакеров. Она предназначена всем, кто хочет попрактиковаться в языке Java, чтобы потом создавать реальные приложения. Эта книга также может рассматриваться как экспресс-курс по объектно-ориентированному программированию, работе с сетями, GUI и XML. При изучении Java вы получите понятие о мощном и практическом подходе к разработке программного обеспечения, начинающемся с глубокого понимания основ языка Java и его интерфейсов API.

При поверхностном рассмотрении Java выглядит как язык C или C++, поэтому вы получите небольшое преимущество в работе с этой книгой, если у вас есть какой-то опыт программирования на одном из этих языков. Если у вас нет такого опыта, не беспокойтесь. Не проводите слишком много синтаксических параллелей между языком Java и C или C++. Во многих отношениях Java ведет себя как такие более динамические языки, как Smalltalk и Lisp. Знание другого объектно-ориентированного языка определенно должно помочь, хотя вам, возможно, придется поменять некоторые представления и отучиться от некоторых привычек. Java значительно проще, чем другие языки, такие как C++ и Smalltalk. Если вы внимательно изучите материал по конкретным примерам и посредством собственных экспериментов, мы думаем, вам понравится эта книга.

Последняя часть этой книги посвящена обсуждению Java в контексте веб-приложений, веб-сервисов и обработки XML, поэтому вы должны быть знакомы с базовыми понятиями о веб-браузерах, сервисах и документации.

Новые разработки

Это издание на самом деле является шестым — усовершенствованным и переименованным — после выхода оригинальной популярной книги Learning Java. С каждым изданием мы заботились не только о добавлении нового материала, охватывающего дополнительные свойства, но и о тщательной проверке и обновлении существующего содержания, чтобы объединить информацию и добавить плоды накопленного за годы фактического опыта к этим страницам. Значи-

тельным изменением в последних изданиях является то, что мы уделили меньшее значение использованию апплетов, отражая их меньшую роль в создании интерактивных веб-страниц в последние годы. С другой стороны, мы значительно расширили информацию о веб-приложениях Java, веб-сервисах и XML, которые являются развитыми технологиями.

Мы рассмотрели все важные свойства выпуска Java Standard Edition (SE) 7, JDK 1.7 и описали нововведения релиза Java 8. Компания Sun (владелец языка Java до Oracle) неоднократно меняла схему наименования за много лет. Sun придумала термин Java 2, чтобы учесть основные новые изменения, представленные в версии Java 1.2, и ввела термин JDK в пользу SDK. С шестым выпуском Sun перепрыгнула с версии Java 1.4 до версии Java 5.0, но дала отсрочку термину JDK и оставила свою систему нумерации. После этого у нас появился язык Java 6, и теперь мы добрались до Java 7 и 8.

Этот выпуск Java демонстрирует зрелый язык с относительно малыми синтаксическими изменениями, но значительными обновлениями интерфейсов API и библиотек. Мы попытались охватить эти новые свойства и обновить каждый пример в этой книге, чтобы отразить не только современные приемы Java, но и стиль.

Новое в этом издании

Это значительно переработанное, наиболее полное и современное издание. Оно включает изменения из выпусков Java 6, 7 и 8, которые произошли с последнего издания этой книги. Новые темы этого издания включают:

- новые свойства языка, в том числе вывод типа в обобщениях и улучшенную обработку исключений и автоматический синтаксис управления ресурсами;
- новые утилиты параллельности, включая фреймворк разветвления-соединения;
- новую NIO Files API, которая позволяет реализацию новых типов доступа к файловой системе в Java;
- новые версии Java Servlets API (3.0) и API веб-сервисов, включая использование нового размещения, основанного на аннотациях и встроенного контейнера веб-сервисов;

- новую версию JAXB (2.2) Java XML Binding, включающую использование новых аннотаций для связывания Java с XML;
- улучшенную настольную интеграцию Swing и улучшение ключевых компонентов Swing, таких как JTable;
- обновленные примеры и анализ во всей книге.

Использование этой книги

Эта книга организована следующим образом.

- Главы 1 и 2 представляют базовый ввод в концепцию Java и учебное пособие для стартового прыжка в Java программирование.
- В главе 3 обсуждаются основные инструменты разработки на Java (компилятор, интерпретатор и упаковщик файлов JAR).
- Главы с 4 по 7 описывают сам язык Java, начиная с основного синтаксиса, а затем обозревают классы и объекты, исключения, массивы, перечисления, комментарии и многое другое.
- Глава 8 описывает базовые и параметризированные типы в Java.
- Глава 9 описывает встроенные возможности потоков и пакет Java Concurrency, который будет особенно интересен для продвинутых программистов.
- Глава 10 описывает текстовую обработку, форматирование, сканирование, строковые утилиты и мощные интерфейсы регулярных выражений API.
- Глава 11 описывает ядро интерфейса API, включая утилиты и коллекции.
- Глава 12 описывает ввод/вывод в Java, потоки, файлы и упаковку NIO.
- Главы 13 и 14 описывают работу с сетями в Java, включая сокеты и NIO, URL и RMI.
- Глава 15 описывает веб-приложения, используя небольшие специальные программы — сервлеты, фильтры на их основе и файлы WAR, а также веб-сервисы.
- Главы с 16 по 21 описывают GUI (графический интерфейс пользователя) разработку с Abstract Window Toolkit (AWT — набор ин-

струментальных средств для абстрактных окон) и Swing, который представляет графический интерфейс пользователя и поддержку изображений.

- Глава 22 описывает архитектуру компонента JavaBeans и знакомит с NetBeans IDE.
- Глава 23 описывает апплеты — небольшие прикладные программы.
- Глава 24 описывает Java API для работы с XML и XSLT, включая XML Schema, валидацию, XPath и XInclude, так же как XML, связанный с JAXB.
- Приложение А описывает использование IDE Eclipse с примерами в книге.
- Приложение Б описывает BeanShell, облегченный язык скриптов для разработки на Java от авторов этой книги.

Если вы похожи на нас, то не станете читать книгу от корки до корки. Если вы и впрямь такие же, как мы, вы обычно вообще не читаете введение.

Однако на случай, если вы увидите это вовремя, вот несколько советов.

- Если вы опытный программист, который должен выучить Java за следующие пять минут, вы, вероятно, ищете примеры. Можете для начала просмотреть мельком главу 2. Если это вас не прельщает, вам стоит хотя бы заглянуть в главу 3, в которой объясняется, как использовать компилятор и интерпретатор, или приложение А, в котором показано, как запустить примеры в IDE Eclipse. С этого вам следует начать.
- Главы с 12 по 15 необходимы, если вас интересует создание продвинутых сетевых или веб-приложений и сервисов. Это одна из наиболее интересных и важных частей Java.
- Главы с 16 по 22 описывают графические функции Java и компоненты архитектуры. Вам следует это прочесть, если вы интересуетесь написанием графических приложений Java или коротких программ-апплетов.
- Глава 24 описывает, как API Java работает с XML, включая SAX, DOM, DTD, XML Schema, и как, используя XSL, выйти в веб-среду. Технология XML становится ключом для кросс-платформенной разработки.

Веб-ресурсы

Существует множество веб-ресурсов с информацией о Java. Официальный сайт Oracle по языку Java — **java.sun.com**; ищите здесь программное обеспечение, обновления и релизы Java. Здесь вы найдете инструментарий JDK, который включает компилятор, интерпретатор и другие инструменты.

Примеры к книге

Вам также следует посетить сайт **github.com/patniemeyer/learningjava**. Здесь вы найдете исходные коды примеров из этой книги.

Условные обозначения, использованные в этой книге

Условные обозначения при помощи шрифта в этой книге довольно просты.

Курсив используется для выделения:

- имен путей, файлов и программ;
- иовых терминов, где они определяются;
- потоков.

Полужирный шрифт используется для выделения:

- веб-адресов, таких как имена доменов и URL-адреса.

`Моноширинный шрифт` используется для выделения:

- любых элементов языка Java, включая имена методов, переменные и имена классов;
- тегов, которые могут появиться в документах HTML или XML;
- ключевых слов, объектов и переменных среды.

`Моноширинный полужирный шрифт` используется для выделения:

- текста, который набирается пользователем в командной строке.

`Моноширинный курсивный шрифт` используется для выделения:

- заменяемых фрагментов кода.



Этот значок обозначает важное примечание к соседнему тексту.



Этот значок обозначает предупреждение, относящееся к соседнему тексту.

В тексте мы указываем пару пустых круглых скобок после имени метода, чтобы отличать методы от переменных и других элементов.

В листинге исходного кода Java мы придерживаемся соглашений по программированию, наиболее часто используемых в сообществе Java. Имена классов начинаются с заглавных букв; имена методов и переменных — со строчных. Все буквы в именах констант в верхнем регистре. Мы не используем подчеркиваний, чтобы разделять слова в длинном названии; придерживаясь распространенной практики, мы пишем с большой буквы отдельные слова (после первого) и соединяем слова вместе. Например: `thisIsAVariable`, `thisIsAMethod()`, `ThisIsAClass` и `THISISACONSTANT`. Также обратите внимание, что мы различаем статические и нестатические методы, когда ссылаемся на них. В отличие от некоторых книг мы никогда не пишем `Foo.bar()`, чтобы подразумевать метод `bar()` объекта `Foo`, если `bar()` не является статическим методом (распараллеливая синтаксис Java в этом случае).

Использование примеров кода

Эта книга призвана помочь вам. В общем, если эта книга включает примеры кода, вы можете использовать их в ваших программах и документации. Вам не нужно связываться с издательством, чтобы получить разрешение, если вы не воспроизводите значительную часть кода. Например, написание программы, которая использует несколько отрывков кода из этой книги, не требует разрешения. Продажа или распространение примеров кода на компакт-диске требует разрешения издательства. Упоминание или цитирование этой книги и приведение примера кода не требует разрешения. Включение значительного количества кода из примеров этой книги в документацию вашего продукта требует разрешения.

Мы ценим ссылки на источник, хотя и не настаиваем на них. Ссылка на источник обычно включает название книги, имя автора, издательство и год выхода книги.

Благодарности

Много людей внесли свой вклад в создание этой книги и в ее воплощение в печатном виде. Прежде всего мы хотим поблагодарить Тима О'Рейли за то, что он дал нам возможность написать эту книгу. Спасибо Майку Лукидесу, редактору, чье терпение и опыт помогли нам начать это путешествие. Спасибо Пауле Фергюсон и Джону Познеру, которые помогали в организационных и редакционных делах. И особая благодарность Дебу Кэмерону, неутомимому редактору этой книги, без которого предыдущие два издания могли остаться незаконченными и наверняка не увидели бы свет. Мы не могли даже и мечтать о более умелой или отзывчивой команде.

Если говорить о заимствованиях, терминология была взята из книги Дэвида Флэнагана (David Flanagan) Java. In a Nutshell (O'Reilly, 2014). Мы также взяли несколько диаграмм иерархии классов из этой книги. Эти диаграммы были основаны на созданных Чарльзом Л. Перкинсом (Charles L. Perkins).

Также благодарим Марка Уоллеса (Marc Wallace) и Стивена Баркетта (Steven Burkett) за чтение оригинальной работы в процессе и за поддержку наших друзей из Университета Джорджа Вашингтона: Брайана О'Коннора (Bryan O'Connor) и Брайена Готтлиба (Brian Gottlieb). Также спасибо Джошу Пеку (Josh Peck), соавтору оригинальной книги Exploring Java. Спасибо всем тем, кто рассматривал или отвечал на них: Дэвиду Флэнагану за обобщения; Генри Вонгу (Henry Wong) за утилиты параллельности; Джиму Эллиотту (Jim Elliott), Марку Лою (Marc Loy) и Брайану Коулу (Brian Cole) за Swing; Джеку Ширази (Jack Shirazi) за NIO; Тиму Будро (Tim Boudreau) за NetBeans; Мартину Эшлиману (Martin Aeschlimann), Джиму Фэрли (Jim Farley) и Джону Норману (John Norman) за Eclipse; Эду Хауланду (Ed Howland) за XML; и Иэну Дарвину (Ian Darwin) за регулярные выражения. Также спасибо Рэю О'Лири (Ray O'Leary), Марио Аквино (Mario Aquino) и Марку Волкманну (Mark Volkmann) за их обзоры. И, в конце концов, спасибо моей красивой жене, Эллен Сонг (Ellen Song), за то, что терпела меня во время всей этой работы.

Глава 1
СОВРЕМЕННЫЙ ЯЗЫК

Огромные перспективы и самые захватывающие возможности для разработчиков программного обеспечения лежат в обуздании силы сетей. Приложения, созданные сегодня, для каких бы целей и аудитории они ни планировались, почти непременно запускаются на машинах, связанных глобальной сетью компьютерных ресурсов. Возрастающая важность сетей предъявляет новые требования к существующим инструментам и заставляет быстро расти список абсолютно новых типов приложений.

Мы хотим иметь программное обеспечение, которое работает стабильно на любой платформе и которое хорошо сочетается с другими приложениями. Мы хотим иметь динамические приложения, которые используют преимущества Всемирной паутины, способны получить доступ к несоизмеримым и распределенным источникам информации. Мы хотим умные приложения, которые могут бродить по сетям вместо нас, разыскивая информацию и служа электронными агентами. Мы уже довольно давно знаем, какое программное обеспечение хотим, но в действительности мы стали получать его только в течение последних нескольких лет.

Исторически проблема заключается в том, что инструментов для создания таких приложений не хватает. Требования к скорости и портативности являются взаимоисключающими, а безопасность в большей степени игнорируется или не понимается. В прошлом действительно мобильные языки были громоздкими, интерпретированными и медленными. Своей популярностью они скорее обязаны высокому уровню функциональности, а не портативности. Многие языки обычно обеспечивали скорость, привязываясь к определенным платформам, так что они решали проблему портативности только наполовину. Существовало даже несколько безопасных языков, но они в основном были ответвлениями портативных, поэтому имели те же проблемы. Язык Java — это современный язык, который работает на трех фронтах: портативность, скорость и безопасность. Поэтому он является доминирующим языком в мире программирования более 15 лет.

Появление Java

Язык программирования Java, разработанный компанией Sun Microsystems под руководством светил Интернета Джеймса Гослинга (James Gosling) и Билла Джоя (Bill Joy), был создан как машинно-независимый язык программирования, который достаточно безопасен для сетевых применений и достаточно мощен, чтобы заменить родной исполняемый код. Язык Java решил проблемы, поднятые выше, и сыграл главную роль в развитии Интернета, результатом чего стало то, что мы видим сейчас.

Изначально энтузиазм в отношении Java был в основном сосредоточен на его возможностях создания встраиваемых приложений для Всемирной паутины, называемых *апплетами*. Но вначале апплеты и другие приложения для графических интерфейсов пользователя (GUI), написанные на Java, были очень ограниченными. Сегодня у Java есть Swing — один из самых изощренных наборов инструментов для создания GUI среди всех языков. Эта разработка позволила языку Java превратиться в популярную платформу для разработки традиционного клиентского прикладного программного обеспечения.

Что еще более важно, язык Java стал главной платформой для веб-приложений и сервисов. Эти приложения используют технологии, включающие Java Servlet API, веб-сервисы Java и много популярных коммерческих серверов приложений и фреймворков Java и серверов и фреймворков с открытым кодом. Портативность и скорость языка Java делают его платформой номер один для современных бизнес-приложений. Серверы языка Java, работающие на платформах Linux с открытым кодом, находятся сегодня в сердце делового и финансового мира.

Эта книга покажет вам, как использовать язык Java для выполнения реальных задач по программированию. В следующих главах мы рассмотрим все от текстовых процессоров до создания сетей, создания многофункциональных клиентских приложений GUI при помощи набора инструментов Swing и легковесных веб-приложений и сервисов.

Происхождение Java

Семена Java были посеяны в 1990 году основателем и главным исследователем компании Sun Microsystems Биллом Джоем. В это вре-

мя компания Sun конкурировала на относительно небольшом рынке рабочих станций, в то время как компания Microsoft начинала доминировать своими более массовыми персональными компьютерами с микропроцессорами Intel. Когда Sun упустила свой шанс в революции персональных компьютеров, Джой отступил в Аспен, Колорадо, для работы в перспективных исследованиях. Он придерживался идеи о выполнении сложных задач с использованием простого программного обеспечения и основал удачно названную компанию Sun Aspen Smallworks. Среди первых членов маленькой команды программистов, собравшихся в Аспене, Джеймса Гослинга запомнят как отца языка Java. Гослинг впервые прославился вначале 80-х как автор Gosling Emacs — первой версии популярного редактора Emacs, который был написан на языке C и работал под Unix. Редактор Gosling Emacs стал популярным, но вскоре его затмила бесплатная версия — GNU Emacs, написанная первым дизайнером Emacs. К тому времени Гослинг перешел к разработке системы NeWS компании Sun, которая какое-то время соперничала с системой X Window System для управления рабочим столом с GUI на основе Unix в 1987 году. Хотя некоторые люди придерживаются точки зрения, что система NeWS предшествовала системе X, NeWS проиграла, потому что компания Sun оставляла ее своей собственностью и не публиковала исходный код, в то время как первые разработчики X сформировали Консорциум X и использовали противоположный подход.

Разработка NeWS обучила Гослинга силе интеграции выразительного языка с GUI оконного типа, ориентированным на сети. Она также научила компанию Sun, что сообщество интернет-программирования, безусловно, отказывается принимать стандарты, защищенные правом собственности, какими бы хорошими они ни были. Семена лицензионной схемы языка Java и открытого кода (пусть даже и не с открытым источником) посеяны неудачей NeWS. Гослинг принес то, чему он научился, в рождающийся проект Билла Джоя в Аспене. В 1992 году работа над проектом привела к основанию корпорации FirsPerson, дочерней корпорации Sun. Ее миссией было вывести Sun в мир потребительской электроники.

Команда компании FirsPerson разрабатывала программное обеспечение для информационной бытовой техники, такой как мобильные телефоны и карманные персональные компьютеры (PDA). Целью было обеспечить переход информации и приложений в реальном времени на дешевые инфракрасные и традиционные сети, основанные на пакетах.

Ограничения в памяти и производительности диктовали компактный и эффективный код. Природа приложений также требовала, чтобы они были безопасными и ясными. Гослинг и члены его команды начали программировать на языке C++, но вскоре были сбиты с толку тем, что язык оказался слишком сложным, громоздким и небезопасным для их задач. Они решили начать с самого начала, и Гослинг начал работать над чем-то, что он принялся «C++ минус минус».

С изобретением Apple Newton (первого карманного компьютера корпорации Apple) стало очевидно, что время карманных персональных компьютеров еще не пришло, поэтому компания Sun переключила усилия FirsPerson на интерактивное телевидение (ITV). Язык программирования, выбранный для телевизионных приставок, стал практически предшественником Java — языком, названным Oak. Даже с его изяществом и возможностью предоставлять безопасную интерактивность язык Oak не смог спасти безнадежное дело ITV в то время. Покупатели его не хотели, и Sun вскоре оставила эту идею.

В то время Джой и Гослинг объединились, чтобы выбрать новую стратегию для своего инновационного языка. Шел 1993 год, и вспышка интереса к Всемирной паутине предоставила им новую возможность. Язык Oak был компактным, безопасным, независимым от архитектуры и объектно-ориентированным. Как оказалось, это также являлось некоторыми из требований к универсальному, понятному Интернету языку программирования. Sun быстро сменила фокус, и после небольших изменений язык Oak стал языком Java.

Становление

Не было бы преувеличением сказать, что язык Java разгорелся как дикий огонь. Даже до его официального выпуска, когда язык Java не был производственным, почти каждая крупная компания отрасли примкнула к нему. Лицензиатами Java были Microsoft, Intel, IBM и почти все основные производители аппаратных средств и программного обеспечения. Однако со всей этой поддержкой язык Java потерпел много неудач и узнал боль перемен в свои первые годы.

Серия нарушений договоров и антитрестовских судебных процессов между компаниями Sun и Microsoft по поводу распространения Java и его использования в браузере Internet Explorer препятствовала применению его в самой распространенной в мире операционной

системе для настольного компьютера — Windows. Затруднительное положение Microsoft и Java также стало одним из очагов большого федерального дела о серьезных противоконкурентных практиках в компании, в результате чего свидетельские показания в суде выявили согласованные усилия гиганта программного обеспечения с целью подорвать Java, демонстрируя несоответствия в их версии языка. В то же время Microsoft представила свой собственный язык C# (C-sharp), основанный на Java, как часть своей инициативы .NET и избавилась от включения Java в Windows. C# стал очень хорошим языком сам по себе, совершенствуясь бóльшим количеством инноваций в последние годы, чем Java.

Но Java продолжает распространяться на самых разнообразных платформах. Когда мы начнем знакомиться с архитектурой Java, вы увидите, что много удивительного в этом языке происходит от автономной среды виртуальной машины, на которой работают приложения Java. Язык Java очень тщательно разработан, так что эта поддерживающая архитектура может быть применена как в программном обеспечении для существующих компьютерных платформ, так и для настроенного аппаратного обеспечения. Применение Java в аппаратном обеспечении используется в некоторых чиповых картах и других встроенных системах. Вы даже можете купить «носимые» устройства, такие как кольца и армейские жетоны, в которых встроены интерпретаторы Java. Применение программного обеспечения Java доступно для всех современных компьютерных платформ, включая портативные компьютерные устройства. Сегодня ответвление платформы Java является основой операционной системы Android от Google, на которой работают миллиарды телефонов и других мобильных устройств.

В 2010-м корпорация Oracle купила Sun Microsystems и стала распорядителем языка Java. Начало было непростым: Oracle судилась с Google из-за использования языка Java в Android и проиграла. В июле 2011-го Oracle выпустила Java SE 7, важный выпуск Java.

Виртуальная машина

Язык Java является скомпилированным и интерпретированным языком. Исходный код Java превращен в простые бинарные инструкции, что больше похоже на машинный код микропроцессора. Как бы то ни было, если источник C или C++ уменьшен до родных инструкций

для определенной модели или процессора, то код Java скомпилирован в универсальный формат — инструкции для *виртуальной машины*.

Скомпилированный *байт-код* Java выполняется интерпретатором времени выполнения. Система поддержки выполнения выполняет все обычные действия процессора аппаратного обеспечения, но делает это в безопасной виртуальной среде. Она выполняет набор стековых инструкций и управляет памятью как операционная система. Она создает простые типы данных и управляет ими, загружает и вызывает вновь ссылаемые блоки кода. Что самое главное, она делает это в соответствии с четко определенной открытой спецификацией, которую может применить любой, кто хочет создать виртуальную машину Java. Вместе виртуальная машина и определение языка представляют полную спецификацию. Нет свойств базового языка Java, которые бы остались неопределенными или зависимыми от внедрения. Например, язык Java специфицирует размеры и математические свойства всех своих простых типов данных, а не оставляет их реализации платформы.

Интерпретатор Java довольно легковесный и компактный, он может быть реализован в любой форме, подходящей для определенной платформы. Интерпретатор может работать как отдельное приложение или может быть внедрен в другое программное обеспечение, такое как веб-браузер. Все сказанное означает, что код Java полностью портативен. Один и тот же байт-код приложения Java может работать на любой платформе, что обеспечивает среду выполнения Java, как показано на рис. 1.1. Вам не нужно производить альтернативные версии вашего приложения для разных платформ и не нужно раздавать исходный код конечным пользователям.

Фундаментальная единица кода Java — *класс*. Как и в других объектно-ориентированных языках, классы являются компонентами приложения, которые содержат исполняемый код и данные. Скомпилированные классы Java распространяются в универсальном бинарном формате, содержащем байт-код Java и другую информацию класса. Классы могут сохраняться отдельно и храниться в файлах или архивах локально или на сетевом сервере. Классы располагаются и подгружаются динамически во время исполнения, когда они необходимы приложению.

В дополнение к платформо-зависимым системам исполнения Java имеет ряд фундаментальных классов, которые содержат архитектурно-зависимые методы. Эти *«родные» методы* служат воротами между виртуальной машиной Java и реальным миром. Они реализованы в первона-

чально скомпилированном языке на платформе хоста и предоставляют доступ нижнего уровня к ресурсам, таким как сеть, система управления окнами и файловая система хоста. Подавляющая часть Java написана на самом языке Java, загруженном с этих базовых элементов, и поэтому портативна. Это базовые инструменты Java, такие как компилятор Java, сетевые библиотеки и библиотеки GUI, которые также написаны на Java и доступны на всех платформах Java точно таким же образом без портирования.

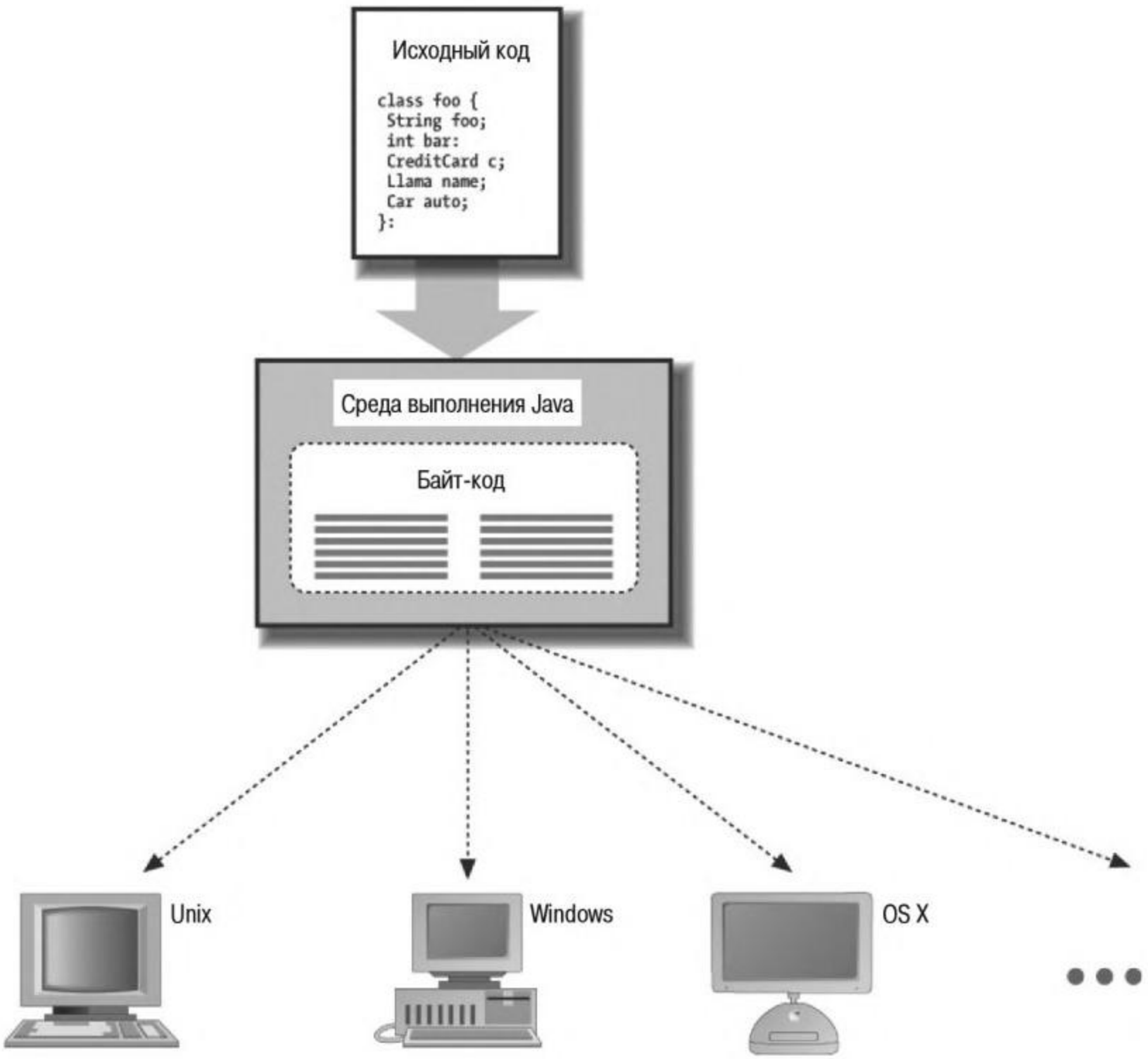

Рис. 1.1. Среда выполнения Java

Исторически интерпретаторы считались медленными, но Java не является традиционным интерпретированным языком. В дополнение к компиляции исходного кода в портативный байт-код Java была тщательно разработана таким образом, что программная реализация системы выполнения может далее оптимизировать свою производительность, компилируя байт-код в оригинальный машинный код на лету. Это на-

зывается компиляция «точно в срок», или динамическая компиляция. С динамической компиляцией код Java может выполняться так же быстро, как оригинальный код, и поддерживать свою транспортабельность и безопасность.

Этот пункт часто непонятен тем, кто хочет сравнить производительность языков. Существует только одна серьезная проблема, от которой страдает компилируемый Java-код во время выполнения ради безопасности и архитектуры виртуальной машины, — проверка границ массива. Все остальное может быть оптимизировано в оригинальный код так же, как при статической компиляции языка. Кроме этого, язык Java содержит больше структурной информации, чем многие другие языки, предусматривая больше типов оптимизации. Также помните, что эти оптимизации могут осуществляться во время выполнения, принимая во внимание реальное поведение и характеристики приложений. Что может быть сделано во время компиляции, чего нельзя сделать лучше во время выполнения? Что ж, кое-чем приходиться поступиться: временем.

Проблема традиционной динамической компиляции — это то, что оптимизация кода занимает время. Итак, динамический компилятор может давать достойные результаты, но может страдать от значительного времени ожидания при запуске приложения. В основном это не имеет большого значения для долго работающих приложений серверной стороны, но является серьезной проблемой для программного обеспечения клиентской стороны и приложений, которые работают на меньших устройствах с ограниченными возможностями. Чтобы решить этот вопрос, технология компилирования Java, называемая HotSpot, использует прием, именуемый *адаптивной компиляцией*. Если вы посмотрите, на что в действительности расходуется время работы программ, окажется, что они тратят почти все время, выполняя относительно небольшую часть кода снова и снова. Участок кода, который повторно выполняется, может быть только малым фрагментом всей программы, но его поведение определяет общую производительность программы. Адаптивная компиляция также позволяет времени выполнения Java использовать преимущества новых типов оптимизации, которая не может быть выполнена в статически компилируемых языках, отсюда утверждение, что Java-код работает быстрее, чем C/C++ в некоторых случаях.

Чтобы получить преимущество от этого факта, технология HotSpot начинает как обычный интерпретатор байт-кода Java, но с одной разницей: она измеряет (профилирует) код во время его выполнения, чтобы

увидеть, какие части кода выполняются многократно. Когда она знает, какие части кода являются критическими для производительности, технология HotSpot компилирует эти части в оптимальный собственный машинный код. Поскольку она компилирует только маленькую часть программы в машинный код, она имеет возможность использовать время, необходимое для оптимизации этих частей. Остальная часть программы может вообще не нуждаться в компиляции — только интерпретации — что сохраняет память и время. В действительности виртуальная машина Java может работать в двух режимах: клиентском и серверном, что определяет, делать акцент на быстром времени старта и сохранении памяти или на скорости выполнения.

В этом месте возникает естественный вопрос: зачем выбрасывать всю эту хорошую профильную информацию каждый раз, когда выключается приложение? Что ж, компания Sun затронула эту тему во время релиза Java 5.0 через использование общих классов и классов только для чтения, которые перманентно хранятся в оптимизированной форме. Это значительно уменьшило и время запуска, и наложение многих Java-приложений на одной машине. Эта технология довольно сложная, но идея простая: оптимизировать части программы, которые должны работать быстро, и не беспокоиться обо всем остальном.

Java в сравнении с другими языками

В своем выборе свойств язык Java опирается на многолетний опыт программирования на других языках. Следует уделить немного времени сравнению Java на высоком уровне с некоторыми другими языками. Это будет полезно для тех из вас, у кого имеется опыт программирования на других языках, и для новичков, которым необходимо хорошо разобраться в теме. В этой книге мы не рассчитываем на то, что вы знаете какой-либо язык программирования, и когда мы ссылаемся на другие языки для сравнения, мы надеемся, что комментарии говорят сами за себя.

Как минимум три колонны необходимы для поддержки универсального языка программирования сегодня: портативность, скорость и безопасность. Рис. 1.2 показывает язык Java в сравнении с несколькими другими языками, которые были популярны, когда он был создан.

Вы, возможно, слышали, что язык Java во многом похож на язык C или C++, но в действительности это только внешнее сходство. Когда вы впервые взглянете на Java-код, вы увидите, что базовый синтаксис по-

хож на C или C++ . Но на этом сходство и заканчивается. Язык Java ни в коем случае не является прямым потомком C или следующим поколением C++. Если вы сравните свойства этих языков, вы увидите, что у Java в действительности больше общего с динамическими языками высшей степени, такими как Smalltalk и Lisp. В действительности реализация Java настолько далека от реализации C++, насколько вы можете себе представить.

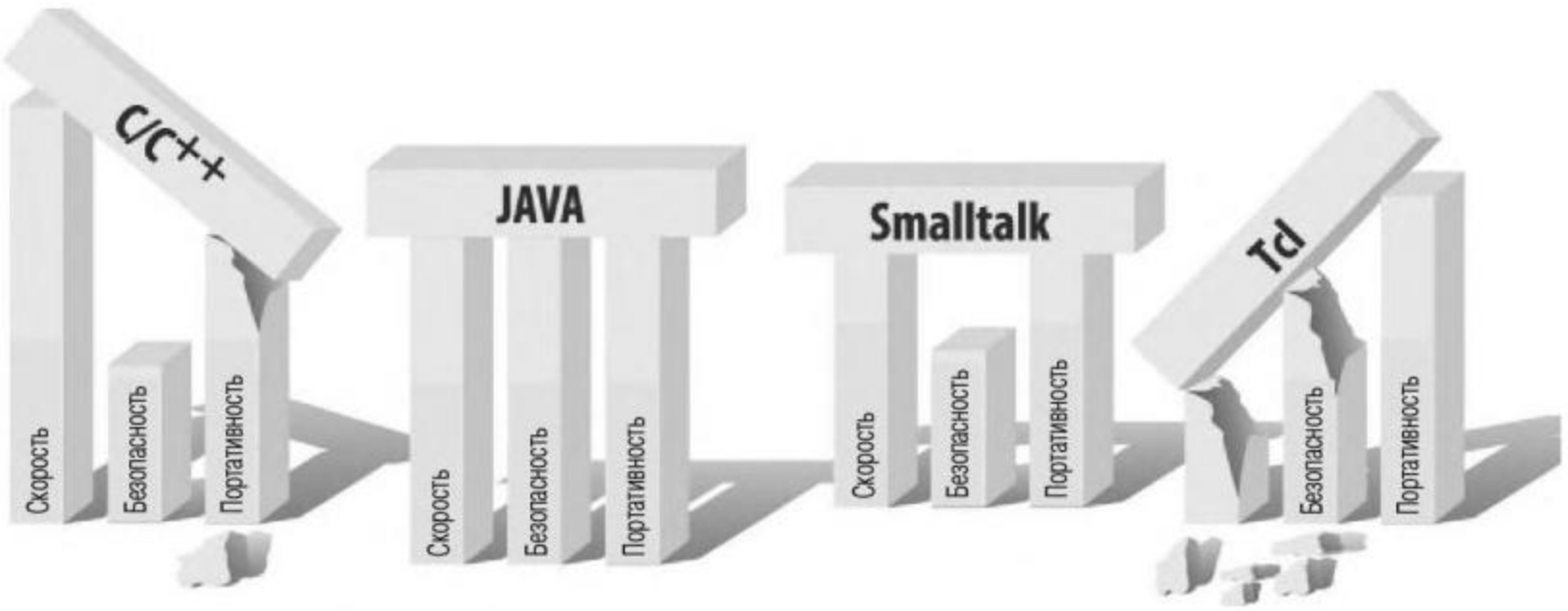

Рис. 1.2. Сравнение языков программирования

Если вы знакомы с современной панорамой языков, вы заметите, что популярный язык C# отсутствует в этом сравнении. C# в целом — это ответ Microsoft на Java, имеющий много преимуществ при поверхностном рассмотрении. При общем дизайне и подходе (например, использование виртуальной машины, байт-код, «песочница» и др.) платформы по существу не отличаются в показателях скорости или характеристиках безопасности. C# теоретически является таким же портативным, как Java, но сегодня он поддерживается на гораздо меньшем количестве платформ. Как и Java, C# много позаимствовал из синтаксиса C, но в действительности является близким родственником динамических языков. Многие разработчики Java считают, что понять C# достаточно легко, и наоборот. Большая часть времени, которое тратится при переходе с одного языка на другой, уходит на изучение стандартной библиотеки.

Поверхностное сходство языков, однако, ничего не стоит. Язык Java берет многое от синтаксиса языка C и C++, так что вы увидите краткие языковые конструкции, включая изобилие фигурных скобок и точек с запятой. Язык Java подписывается под философией C, заключающейся в том, что хороший язык должен быть компактным, другими словами, он должен быть достаточно легковесным и регулярным, так что про-

граммист сможет сразу запомнить все его возможности. Как и в случае с расширяемостью библиотек языка C, пакеты с классами языка Java могут добавляться в центральные компоненты языка для расширения его словаря.

Язык C успешен, поскольку предоставляет программную среду со значительным содержанием свойств, с высокой производительностью и приемлемой долей портативности. Язык Java также старается балансировать между функциональностью, скоростью и портативностью, но совсем другим образом. Язык C выдает функциональность за портативность, язык Java изначально выдавал скорость за портативность. Язык Java также решает проблемы безопасности, которые язык C не решает (хотя в современных системах многие из этих проблем сейчас решаются операционной системой и аппаратным обеспечением).

В самом начале, еще до динамической и адаптивной компиляции, язык Java был медленнее, чем статически компилируемые языки, и злые языки постоянно повторяли, что он никогда не достигнет достойного уровня. Но, как мы сказали в предыдущем разделе, производительность Java сейчас сравнима с C или C++ в эквивалентных задачах, и эта критика в общем сошла на нет. Движок видеоигры с открытым кодом Quake2, созданной ID Software, был переписан на Java. Если язык Java является достаточно быстрым для военной игры от первого лица, то он определенно достаточно быстр для бизнес-приложений.

Языки сценариев, такие как Perl, Python, и Ruby, очень популярны. Нет причин, чтобы язык сценариев не подходил для безопасных сетевых приложений. Но многие скриптовые языки плохо подходят для крупномасштабного программирования. Привлекательным в скриптовых языках является то, что они динамические и являются мощным инструментом для быстрой разработки. Некоторые языки сценариев, такие как Perl, также предоставляют мощные инструменты для задач по обработке текста, которые многие универсальные языки находят громоздкими. Скриптовые языки также очень портативны, хотя и на уровне исходного кода.

JavaScript не стоит путать с Java, так как это объектно-ориентированный скриптовый язык, изначально разработанный Netscape для веб-браузеров. Он является «постоянным жителем» веб-обозревателя для динамических, интерактивных приложений. Название JavaScript происходит от его интеграции с Java и некоторых общих черт, но сходство в действительности здесь и заканчивается. Хотя приложения на JavaScript вне браузера есть, он так по-настоящему и не привился как

скриптовый язык. Для большей информации по JavaScript читайте книгу: «JavaScript и jQuery. Исчерпывающее руководство» (2-е издание) Дэвида Макфарланда (Эксмо, 2012).

Проблема языков сценариев заключается в том, что структура программ и типы данных в них довольно бессистемны. Большинство скриптовых языков (за колеблющимся исключением Python и последних версий Perl) не являются объектно-ориентированными. У них также упрощенные системы типов, и в основном они не предусматривают сложных масштабов переменных и функций. Эти характеристики делают их менее подходящими для создания больших модульных приложений. Скорость — еще одна проблема скриптовых языков; высокоуровневая, зачастую с интерпретацией источника, природа этих языков делает их довольно медленными.

Защитники отдельных скриптовых языков не согласятся с некоторыми из этих обобщений и, без сомнения, будут правы в ряде случаев. Скриптовые языки улучшились за последние годы — особенно язык JavaScript, на производительность которого затрачено большое количество ресурсов. Но фундаментальная уступка неоспорима: скриптовые языки появились как свободная, менее структурированная альтернатива системным языкам программирования и в основном неидеальны для больших или сложных проектов по различным причинам, по крайней мере сегодня.

Java предлагает некоторые существенные преимущества скриптового языка: он в высшей степени динамичен вместе с добавленными плюсами языков низшего уровня. У Java мощный интерфейс Regular Expression API, который соперничает с языком Perl в работе с текстом и свойствах языка, ускоряющими программирование с помощью библиотек, списков переменных аргументов, статического импорта методов и других синтаксических «плюшек», которые делают его более сжатым.

Инкрементная разработка с объектно-ориентированными компонентами в сочетании с простотой Java делает разработку приложений быстрой, а их изменение — легким. Исследования показали, что разработка на языке Java осуществляется быстрее чем на C или C++, строго основанных на свойствах языка*. Java также идет с большой базой

* Для примера см. G. Phipps, “Comparing Observed Bug and Productivity Rates for Java and C++,”Software — Practice & Experience, volume 29, 1999.

стандартных классов ядра для общих задач, таких как разработка GUI и управление сетевыми связями. Но вместе с этими свойствами язык Java имеет масштабируемость и преимущества разработки ПО, присущие более статическим языкам. Он предоставляет безопасную структуру для создания высокоуровневых структур (и даже других языков).

Как мы уже сказали, по конструкции Java похож на Smalltalk и Lisp. Однако эти языки использовались больше как аппараты для научных исследований, чем для создания крупномасштабных систем. Одной из причин является то, что у этих языков никогда не было стандартных портативных связей с сервисами операционных систем, таких как стандартная библиотека C или центральные классы Java. Smalltalk компилируется в интерпретированный формат байт-кода и может интерпретироваться в собственный код на лету, как Java. Но Java улучшен в конструкции за счет использования верификатора байт-кода для обеспечения корректности скомпилированного Java-кода. Этот верификатор дает Java преимущество в производительности перед Smalltalk, поскольку Java-код требует меньших проверок рабочего цикла программы. Верификатор байт-кода Java также помогает с решением проблем безопасности — то, с чем не работает Smalltalk.

В этой главе мы совершим краткий обзор языка Java. Мы объясним, что нового и не очень в Java и почему.

Безопасное проектирование

Вы, должно быть, много слышали о том, что язык Java разработан как безопасный язык. Но что подразумевается под безопасностью? Безопасность от чего или кого? Средства безопасности, которые привлекают много внимания к Java, — это те свойства, которые делают возможными новые типы динамически портативного программного обеспечения. Язык Java обеспечивает несколько уровней защиты от опасно поврежденного кода, а также других вредных явлений, таких как вирусы и троянские кони. В следующем разделе мы посмотрим на то, как архитектура виртуальной машины определяет безопасность кода перед его запуском и как *загрузчик классов* (механизм загрузки интерпретатора Java в байт-коде) строит стену вокруг ненадежных классов. Эти средства составляют основу для формирования высокоуровневой политики безопасности, с помощью которой можно разрешать или запрещать разного рода действия для всех приложений в рамках виртуальной машины.

Однако в этом разделе мы рассмотрим некоторые свойства языка программирования Java. Возможно более важной, чем средства безопасности Java, хотя часто незамеченной, является безопасность, которую предоставляет Java при решении распространенных проблем дизайна и программирования. Задача языка Java — быть как можно более защищенным как от простых ошибок, которые мы делаем сами, так и от тех, которые попадают к нам с унаследованным ПО. Целью Java было оставить язык простым, предоставить инструменты, которые продемонстрируют свою полезность и позволят пользователю создавать при необходимости более сложные объекты на высшем уровне языка.

Упрощение, упрощение, упрощение

В Java простота является правилом. Поскольку язык Java начинался с чистого листа, ему удалось избежать свойств, которые оказались запутанными или противоречивыми в других языках. Например, Java не допускает перегрузку операций, определенных программистом (которая в некоторых языках позволяет программистам переопределять значение базовых символов, таких, как + и –). У языка Java нет препроцессора исходного кода, так что в нем нет таких вещей, как макросы, инструкции `#define` или компиляция условного источника. Эти конструкции существуют в других языках в основном для поддержки зависимостей платформ, так что в этом смысле они не нужны в Java. Условная компиляция также широко используется для отладки, но сложная оптимизация рабочего цикла Java и такие свойства, как *утверждения*, решают проблему более элегантно (мы рассмотрим их в главе 4).

Язык Java предоставляет четкую структуру *пакетов* для организации файлов классов. Система пакетов позволяет компилятору управлять некоторой функциональностью стандартной утилиты `make` (инструмент для создания выполняемого кода из исходного кода). Компилятор также может работать непосредственно с компилированными файлами Java, потому что сохраняется вся информация типа; нет необходимости во внешних «заголовочных» файлах, как в C/C++. Все это означает, что Java-коду нужно читать меньше контекста. В действительности иногда вы можете обнаружить, что быстрее посмотреть на исходный код Java, чем обращаться к документации класса.

Язык Java также по-своему подходит к некоторым структурным свойствам, которые были проблемными в других языках. Например,

Java поддерживает только иерархию классов единичного наследования (у каждого класса может быть только один «родительский» класс), но позволяет многочисленное наследование интерфейсов. *Интерфейс* как абстрактный класс в C++ устанавливает поведение объекта без определения его выполнения. Это очень мощный механизм, позволяющий разработчику определить «контракт» поведения объекта, который может использоваться и упоминаться независимо от частной реализации объекта. Интерфейсы в Java исключают потребность в множественном наследовании классов и связанные с этим проблемы.

Как вы увидите в главе 4, Java достаточно простой и изящный язык программирования, и в этом в первую очередь заключается его притягательная сила.

Безопасность типов и связывание методов

Один из атрибутов языка — это вид *проверки типов*, который он использует. Как правило, языки распределяются по категориям как *статические* и *динамические*, что относится к количеству информации о переменных, известных во время компиляции, против того, что известно во время работы приложения.

В строго статически типизированных языках, таких как C и C++, типы данных словно выбиты на камне, когда компилируется исходный код. Компилятор получает преимущество от этого, поскольку имеет достаточно информации для отлова многих типов ошибок перед выполнением кода. Например, компилятор не позволит вам хранить значение с плавающей точкой в переменной целого типа. Код тогда не нуждается в проверке типов во время выполнения, поэтому он может быть компилирован, чтобы стать компактным и быстрым. Но языки со статическими типами являются негибкими. Они не поддерживают библиотеки так же естественно, как языки с динамической проверкой типов, и делают невозможным для приложения безопасный импорт новых типов данных, когда оно выполняется.

Для сравнения: такой динамический язык, как Smalltalk или Lisp, имеет исполняемую систему, которая управляет типами объектов и производит необходимую проверку типов, пока приложение выполняется. Эти типы языков позволяют более сложное поведение и во многих аспектах более мощные. Однако в основном они медленнее, менее безопасны и тяжелы в отладке.

Разница языков похожа на разницу между автомобилями*. Языки со статическими типами, такие как C++, аналогичны спортивным машинам: довольно безопасны и быстры, но пригодны, только если вы едете по хорошо асфальтированной дороге. Высокодинамические языки, такие как Smalltalk, более похожи на внедорожники: они дают больше свободы, но могут быть немного неуклюжими. Может быть весело (и иногда быстрее) с грохотом проехать через лесную глушь, но вы также можете застрять в яме или быть растерзанным медведями.

Другой атрибут языка — это то, как он связывает вызов методов с их определениями. В таких статических языках, как C или C++, определения методов обычно связываются во время компиляции, если программист не укажет иное. Языки, подобные Smalltalk, с другой стороны, называются *языками с отложенным связыванием*, поскольку они локализируют определения методов динамически во время прогона. Раннее связывание важно из соображений выполнения; приложение может работать без затрат на поиск методов во время прогона. Но отложенное связывание более гибкое. Оно также необходимо в объектно-ориентированном языке, где новые типы могут подгружаться динамически, и только система выполнения может определять, какой метод запускать.

Язык Java предоставляет некоторые преимущества C++ и Smalltalk; это язык со статическими типами и отложенным связыванием. Каждый объект в Java имеет четко определенный тип, который известен во время компиляции. Это означает, что компилятор Java может выполнять такой же тип статических проверок типа и использования анализа, как C++. В результате вы не можете связать объект с неправильным типом переменной или несуществующие методы с объектом. Компилятор Java идет еще дальше и не дает вам использовать неинициализированные переменные и создавать недостижимые инструкции (смотрите главу 4).

Однако Java в полной мере является языком с динамическим выполнением. Система выполнения Java следит за всеми объектами и делает возможным определение их типов и связей во время выполнения. Это означает, что вы можете инспектировать объект во время выполнения, чтобы определить, что это за объект. В отличие от языка C или C++ смена одного объекта на другой отмечается системой выполнения и возможно использование новых типов динамически загружаемых объектов со степенью типа безопасности. И поскольку Java — язык с отложенным свя-

* Автором аналогии с машинами является Маршал П. Кляйн, автор часто задаваемых вопросов C++.

зыванием, подкласс всегда может переопределить методы в его родительском классе, даже если этот подкласс загружен во время выполнения.

Инкрементная разработка

Язык Java переносит все типы данных и информацию о сигнатуре метода с исходного кода в форму компилированного байт-кода. Это означает, что классы Java могут разрабатываться инкрементно. Ваш собственный исходный код Java может также безопасно компилироваться классами из других источников, которые ваш компилятор никогда не видел. Другими словами, вы можете писать новый код, ссылающийся на файлы бинарных классов без потери типов безопасности, которые вы получаете от исходного кода.

Язык Java не страдает от проблем «нестабильных базовых классов». В таких языках, как C++, применение базовых классов может быть эффективно заморожено, поскольку у него много производных классов; изменение базовых классов может потребовать рекомпиляции всех производных. Это особенно сложная проблема для разработчиков библиотек классов. Пока класс управляет действующей формой его оригинальной структуры, он может эволюционировать без нарушения других классов, производных от него или использующих его.

Управление динамической памятью

Некоторые из самых важных отличий между Java и такими языками более низкого уровня, как C и C++, связаны с тем, как Java управляет памятью. Язык Java устраняет случайные «указатели», которые могут ссылаться на произвольные области памяти, и добавляет сборку мусора для объектов и массивы высокого уровня к языку. Эти свойства устраняют проблемы безопасности, портативности и оптимизации, непреодолимые другим путем.

Сборка мусора спасла бесчисленное количество программистов от одного большого источника ошибок программирования на языках C и C++: эксплицитного распределения и освобождения памяти. В дополнение к управлению объектами памяти системы выполнения, система выполнения Java отслеживает все ссылки на этот объект. Когда объект больше не используется, Java автоматически убирает объект из памяти. Вы можете по большей части просто игнорировать объекты,

которые вы больше не используете, с уверенностью, что интерпретатор уберет их в нужное время.

Язык Java использует сложный сборщик мусора, который работает на фоне, то есть большая часть сборки мусора происходит в нерабочее время, между паузами ввода/вывода, щелчками мышью или нажатием клавиш. Более продвинутые системы выполнения, такие как HotSpot, имеют более развитую сборку мусора, которая умеет различать шаблоны использования объектов (например, кратковременные от долговременных) и оптимизировать их сборку. Время выполнения Java теперь может настраиваться автоматически для оптимального распределения памяти для разных типов приложения в зависимости от их поведения. С этим типом профилирования времени выполнения автоматическое управление памятью может осуществляться более быстро, чем наиболее тщательно управляемые программистами ресурсы, — то, во что программисты старой школы до сих пор слабо верят.

Мы сказали, что у Java нет указателей. Собственно говоря, это утверждение верно, но оно также и обманчиво. *Ссылки* — вот что предлагает Java — безопасный тип указателей. Ссылка — это остро типизированный идентификатор объекта. Все объекты в Java, за исключением простых числовых типов, доступны через ссылки. Вы можете использовать ссылки для всех обычных типов структур данных, которые программист на языке C привык создавать указателями, — таких как связанные ссылки, древовидные схемы и т. д. Единственное отличие в том, что со ссылками это делается безопасным путем.

Другое важное отличие между ссылкой и указателем — это то, что вы не можете играть в игры со ссылками (осуществлять арифметику указателей), чтобы изменить их значения; они могут указывать только на определенные объекты или элементы массива. Ссылка — это элементарная вещь; вы не можете управлять значением ссылки иначе, чем назначив ее объекту. Ссылки передаются значением, и вы не можете сослаться на объект более чем через один косвенный уровень. Защита ссылок — один из самых фундаментальных аспектов безопасности. Это означает, что код Java должен играть по правилам; он не может считываться в местах, где не следует, и обходить правила.

Ссылки Java могут указывать только на типы классов. Нет никаких указателей к методам. Некоторые жалуются на их отсутствие, но вы обнаружите, что многие задачи, которые требуют указателей к методам, могут выполняться более чисто при использовании интерфейсов

и классов адаптеров. Мы также должны заметить, что у Java есть сложный интерфейс Reflection API, который в действительности позволяет вам ссылаться и запускать индивидуальные методы. Однако это не является стандартным путем. Мы обсудим отражение в главе 7.

В завершение мы должны отметить, что массивы в Java являются подлинными объектами первого класса. Они могут динамически назначаться и присваиваться как другие объекты. Массивы знают свои размеры и типы и, хотя вы не можете напрямую определить или создать подклассы для классов массива, у них есть четко определенные отношения наследования, основанные на отношениях их базовых типов. Наличие истинных массивов в языке уменьшает нужду в арифметике указателей, которая используется в C или C++.

Обработка ошибок

Корневые каталоги Java находятся в сетевых устройствах и встроенных системах. Для таких приложений очень важно иметь надежное и программируемое управление ошибками. У Java есть мощный механизм обработки исключений, что-то наподобие новых реализаций в языке C++. Исключения обеспечивают более естественный и элегантный метод управления ошибками. Исключения позволяют отделить код управления ошибками от обычного кода, что способствует появлению более чистых и надежных приложений.

Когда возникает исключение, оно вызывает переход выполнения программы с обычного потока на запланированный блок кода «отлова». Исключение влечет за собой объект, содержащий информацию о ситуации, которая вызвала исключение. Компилятор Java требует, чтобы метод объявлял исключения, которые он генерирует, или отлавливал и сам имел дело с ними. Это продвигает информацию об ошибке на тот же уровень важности, что и аргумент, и возвращает типы методов. Как программист Java, вы точно знаете, с какими условиями исключений вам необходимо иметь дело, и у вас есть помощь компилятора в написании правильного программного обеспечения, которое не оставит их необработанными.

Потоки

Современные приложения требуют высокой степени параллелизма. Даже очень целенаправленное приложение может иметь сложный ин-

терфейс пользователя, что требует параллельных операций. В то время как машины становятся быстрее, пользователи становятся более требовательными ко времени, которое тратится на ожидание. Потоки обеспечивают эффективную многопроцессорную обработку и распределение задач между клиентом и серверными приложениями. Язык Java делает потоки легкими в использовании, потому что их поддержка встроена в язык.

Параллелизм хорош, но это в большей степени относится к программированию с потоками, чем просто к выполнению множества задач одновременно. В большинстве случаев потоки должны быть *синхронизированы* (координированы), что может быть сложно без ясной языковой поддержки. Язык Java поддерживает синхронизацию, основанную на модели *наблюдение* и *условие* — тип системы «замка и ключа» для доступа к ресурсам. Ключевое слово синхронизирует объявленные методы и блоки кода для безопасного упорядоченного доступа внутри объекта. Также есть простые, примитивные методы для ясного ожидания и передачи сигналов между потоками, заинтересованными в одном объекте.

У Java также есть пакет параллельной обработки, который предоставляет мощные утилиты, адресующиеся к общим шаблонам в многопоточном программировании, такие как пул потоков, координация задач и сложное связывание. С добавлением пакета параллельной обработки и связанных утилит язык Java предоставляет некоторые из наиболее передовых утилит любого языка, связанного с потоками.

Хотя некоторым разработчикам, возможно, никогда не доведется писать многопоточный код, обучение программированию потоками является важным слагаемым в мастерстве программирования на Java и тем, что все разработчики должны понять. Мы рассмотрим эту тему в главе 9.

Масштабируемость

На низшем уровне программы Java состоят из *классов*. Классы были задуманы как маленькие модульные компоненты. Над классами Java предоставляет *пакеты*, слой структуры, которая группирует классы в функциональные единицы. Пакеты предоставляют соглашения о наименовании для организации классов и второй слой организации контроля над видимостью переменных и методов в приложениях Java.

Внутри пакета класс является либо публично видимым, либо защищенным от внешнего доступа. Пакеты формируют другой тип границ,

который близок к уровню приложения. Оно пригодно для создания компонентов многоразового использования, работающими вместе в системе. Пакеты также помогают в создании масштабируемого приложения, которое может расти, не становясь птичьим гнездом для сильно связанного кода.

Безопасность реализации

Одно дело — создать язык, который не даст вам навредить самому себе; другое — создать язык, благодаря которому другие не будут приносить вред вам.

Инкапсуляция — это принцип сокрытия данных и поведения внутри класса; это важная часть объектно-ориентированного проектирования. Она помогает вам писать чистое, модульное программное обеспечение. Во многих языках, однако, видимость элементов данных — просто часть связи между программистом и компилятором. Это вопрос семантики, а не утверждение о действительной безопасности данных в контексте выполнения программной оболочки.

Когда Бьёрн Страуструп выбрал ключевое слово private для обозначения скрытых членов классов в C++, скорее всего, он думал о защите разработчика от беспорядочных деталей кода другого разработчика, а не о проблемах защиты классов и объектов этого разработчика от атак чьих-то вирусов и троянских коней. Произвольный подсчет и арифметика указателей в языке C или C++ делают обычным делом нарушение доступов к классам без нарушения правил языка. Посмотрите на следующий код:

```
// код C++
class Finances {
private:
char creditCardNumber[16];
...
};
main() {
Finances finances;
// изобретение указателя, чтобы заглянуть в класс
char *cardno = (char *)&finances;
printf("Card Number = %.16s\n", cardno);
}
```

В этом фрагменте на языке C++ мы написали некий код, который нарушает инкапсуляцию класса `Finances` и вытягивает некую секретную информацию. Такой тип махинаций — злоупотребление нетипизированным указателем — невозможен в Java. Если этот пример кажется нереалистичным, обратите внимание на то, как он важен для защиты базовых (системных) классов выполняемой программы от подобных видов атак. Если ненадежный код может повредить компоненты, которые предоставляют доступ к реальным ресурсам, таким как файловая система, сеть или оконная система управления окнами, у него действительно есть все шансы украсть номера вашей кредитной карточки.

Если приложение Java предназначено для динамической загрузки кода из ненадежного источника во Всемирной паутине и запуска его параллельно приложениями, которые могут содержать конфиденциальную информацию, защита должна стать очень глубокой. Модель безопасности Java оборачивает импортированные классы в три уровня защиты, как показано на рис. 1.3.

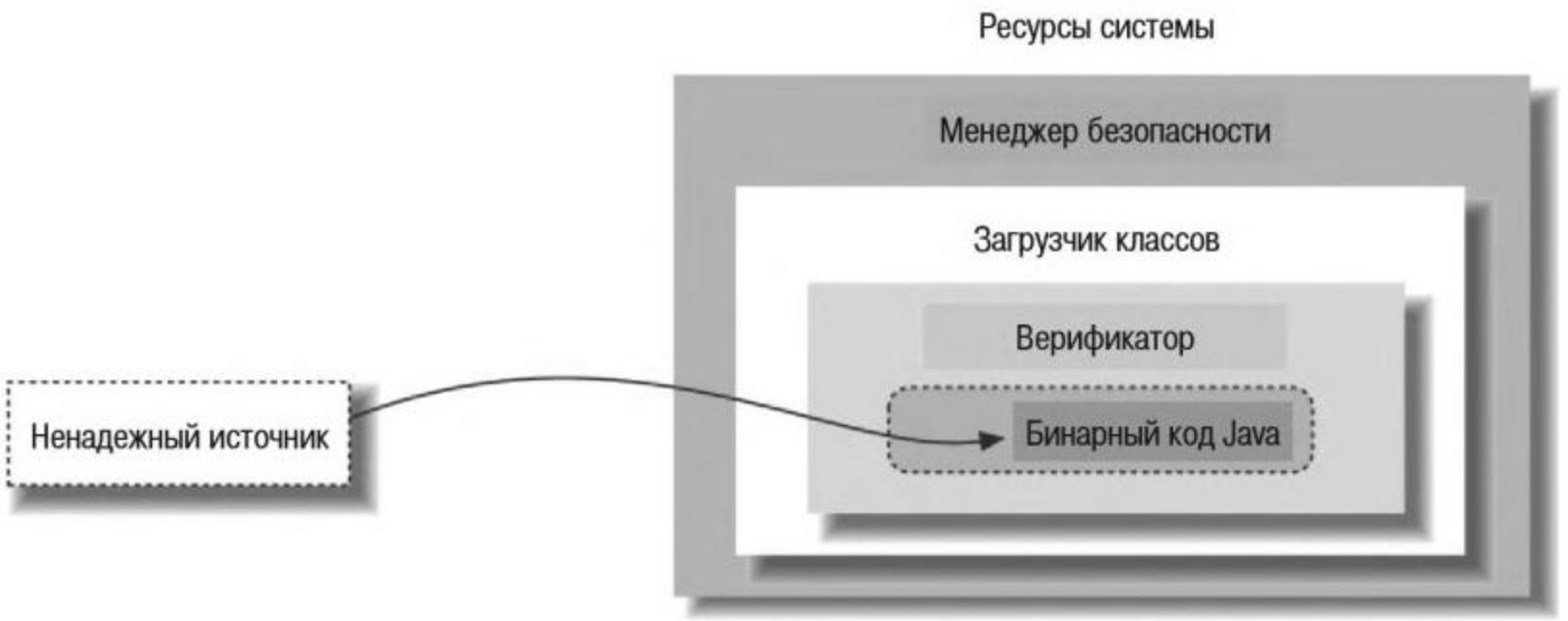

Рис. 1.3. Модель безопасности Java

Снаружи решение безопасности на уровне приложения производится диспетчером безопасности совместно с гибкой политикой безопасности. Диспетчер безопасности контролирует доступ к системам ресурсов, таким как файловые системы, сетевые порты и системы управления окнами. Диспетчер безопасности полагается на возможность загрузчика классов защитить базовую систему классов. Загрузчик классов обрабатывает загружаемые классы отдельно от локального хранилища или сети. На внутреннем уровне вся безопасность системы в конечном счете лежит на верификаторе Java, что гарантирует целостность входящих классов.

Верификатор байт-кода Java является фиксированной частью системы выполнения Java. Загрузчики классов и диспетчеры безопасности (или, если быть более точными, *политика безопасности*), однако, являются компонентами, которые могут выполняться по-разному в разных приложениях, таких как серверы или веб-браузеры. Все эти три компонента должны функционировать должным образом, чтобы обеспечивать безопасность среды Java.

Верификатор

Первая линия защиты языка Java — *байт-кодовый верификатор*. Верификатор считывает байт-код перед его запуском и убеждается, что он работает хорошо и подчиняется общим правилам языка Java. Надежный компилятор Java не производит код, который ведет себя по-другому. Однако какой-то злой человек может преднамеренно собрать плохой байт-код Java. Работа верификатора заключается в том, чтоб обнаружить это.

После того как код был проверен, он считается застрахованным от невнимательности или злоумышленных ошибок. Например, проверенный код не может изобрести ссылки или нарушить разрешения к доступу к объекту (как в примере с кредитными карточками). Он не может выполнять нелегальных приведений типов или использовать объект не предусмотренными путями. Он даже не может вызвать определенные типы внутренних ошибок, такие как переполнение или опустошение внутреннего стека. Эти фундаментальные гарантии лежат в основе безопасности Java.

Вы, возможно, поинтересуетесь, не подразумевается ли этот тип безопасности в большинстве интерпретируемых языков? Что ж, хотя и правда вряд ли сможете разрушить интерпретатор Basic поддельной строчкой кода Basic, помните, что защита во многих интерпретируемых языках происходит на высоком уровне. Вероятно, эти языки имеют тяжеловесные интерпретаторы, которые выполняют большую часть работы, поэтому они непременно более медленные и громоздкие. В то же время байт-код Java является относительно простым набором инструкций низкого уровня. Способность статически проверять байт-код Java перед выполнением позволяет интерпретатору Java работать на полной скорости с полной безопасностью после, без дорогих проверок времени выполнения. Это было одним из базовых нововведений Java.

Верификатор — это тип математического «доказывания теорем». Он проходит через байт-код Java и применяет простые, индуктивные правила для выявления определенных аспектов того, как себя поведет байт-код. Такой тип анализа возможен, поскольку компилируемый байт-код Java содержит намного больше информации о типе, чем объектный код другого языка этого типа. Байт-код также должен следовать нескольким дополнительным правилам, которые упрощают его поведение. Во-первых, большинство инструкций байт-кода работает только с индивидуальными типами данных. Например, в стековых операциях есть отдельные инструкции для ссылок объекта и для каждого из числовых типов в Java. Проще говоря, для перемещения каждого типа значения в локальную переменную и из нее инструкции разные.

Во-вторых, тип объекта, который получится в результате любой операции, всегда известен заранее. Никакая операция байт-кода не уничтожает переменную и не производит более одного возможного типа переменных на выходе. В результате всегда можно взглянуть на следующую инструкцию и ее операнды и узнать тип значения, который получится в результате.

Поскольку операция всегда производит известный тип, можно определить типы всех элементов стека и локальные переменные в любой момент в будущем, посмотрев на исходное состояние. Коллекция всей этой информации типов в любое предоставленное время называется *состоянием типов* стека; это то, что Java старается проанализировать перед запуском приложения. Язык Java ничего не знает о действительных значениях стека и элементах переменных в данное время; он только знает, что это за элементы. Однако этой информации достаточно, чтобы задействовать правила безопасности и удостовериться в том, что объекты не управляются нелегально.

Чтобы сделать реальным анализ типа состояний стека, Java налагает дополнительное ограничение на то, как выполняются инструкции байт-кода Java: все пути к одной точке в коде должны достигаться одним и тем же состоянием типа.

Загрузчики классов

Язык Java добавляет второй уровень защиты при помощи *загрузчиков классов*. Загрузчик класса ответственен за доставку байт-кода для классов Java в интерпретатор. Каждое приложение, которое загружает

классы из сети, должно использовать загрузчик класса, чтобы справиться с этой задачей.

После того как класс загружен и прошел через верификатор, он остается связанным со своим загрузчиком классов. В результате классы эффективно секционируются в отдельные пространства имен, основанные на их происхождении. Когда загрузчик класса ссылается на другое имя класса, местоположение нового класса предоставляется исходным загрузчиком класса. Это означает, что классы, полученные из специфического источника, ограничены до взаимодействия только с другими классами, полученными из того же места. Например, веб-браузер с запущенным приложением Java может использовать загрузчик классов, чтобы создать отдельное пространство для всех классов, загруженных с определенного URL-адреса (единого указателя ресурсов). Сложная система безопасности, основанная на криптографически обозначенных классах, также может быть применена с использованием загрузчиков классов.

Поиск классов всегда начинается с тех, что встроены в систему Java. Эти классы загружаются из мест, определяемых *путями к классу* интерпретатора Java (смотрите главу 3). Классы в пути к классу загружаются системой только однажды и не могут быть заменены. Это означает, что приложение не может заменить фундаментальную систему классов своей собственной версией, которая изменит их функциональность.

Диспетчеры безопасности

Диспетчер безопасности отвечает за принятие решений по безопасности на уровне приложения. Диспетчер безопасности является объектом, который может быть установлен приложением для ограничения доступа к ресурсам системы. К диспетчеру безопасности обращаются каждый раз, когда приложение пытается получить доступ к таким элементам, как файловые системы, сетевые порты, внешние процессы и система управления окнами; диспетчер безопасности может разрешить и запретить запрос.

Диспетчеры безопасности, прежде всего, ценны для приложений, которые выполняют ненадежный код как часть своей нормальной работы. Например, веб-браузеры с запущенным приложением Java способны запускать апплеты, которые могут быть получены от ненадежных источников через Интернет. В таком браузере необходимо установить диспетчер безопасности прежде всего. Затем он ограничивает типы доступа, разрешенные после установки. Это позволяет приложению наложить эффек-

тивный уровень доверия перед запуском случайного фрагмента кода. И когда диспетчер безопасности установлен, его нельзя заменить.

Диспетчер безопасности работает совместно с инспектором доступа, что позволяет вам применять политику безопасности на высоком уровне редактированием декларативного файла политики безопасности. Политика доступа может быть простой или сложной как специфический гарант безопасности приложения. Иногда достаточно просто запретить доступ ко всем ресурсам или общим категориям услуг, таким как файловая система или сеть. Но также возможно принимать сложные решения, основанные на информации высокого уровня. Например, веб-браузеры с запущенным приложением Java могут использовать политику доступа, которая позволяет пользователям определять, насколько можно доверять приложению или которая разрешает или запрещает доступ к специфическим ресурсам от случая к случаю. Конечно, это предполагает, что браузер может определить, какому приложению можно доверять. Скоро мы обсудим, как эта проблема решается через подпись кода.

Целостность диспетчера безопасности основана на защите, предоставленной на низких уровнях модели безопасности Java. Без гарантий, предоставленных верификатором и загрузчиком классов, утверждения о высоком уровне безопасности ресурсов системы являются бессмысленными. Безопасность, обеспеченная верификатором байт-кода Java, означает, что интерпретатор не может быть поврежден или разрушен и что Java-код должен использовать компоненты по назначению. Это в свою очередь означает, что загрузчик классов может гарантировать, что приложение использует системные классы ядра Java и эти классы являются единственным способом доступа к базовым системным ресурсам. С такими ограничениями можно централизовать контроль над этими ресурсами на высоком уровне, с диспетчером безопасности и определяемой пользователем политикой.

Приложение и безопасность на уровне пользователя

Между обладанием достаточной силой, чтобы делать что-то полезное, и обладанием силой делать, что тебе хочется, тонкая граница. Язык Java предоставляет основу для безопасной среды, в которой небезопасный код может быть отправлен в карантин, управляем и безопасно исполняем. Однако, хотя вы удовлетворены тем, что этот код содержится в маленькой черной коробочке и работает только для собственной

пользы, вам придется предоставить ему доступ хоть к какой-то системе ресурсов, чтобы он был полезным. Каждый тип доступа несет с собой определенный тип и выгоду. Например, в среде веб-браузеров преимущества выдачи доступа к вашей системе управления окнами неизвестному (ненадежному) апплету состоят в том, что он сможет показать информацию и позволить вам взаимодействовать в полезном режиме. Связанный с этим риск состоит в том, что апплет вместо этого может показать что-то бесполезное, раздражающее или обидное.

С одной стороны, простой акт запуска приложения дает ему ресурс — время вычисления, который оно может хорошо использовать или легкомысленно прожечь. Тяжело предотвратить трату вашего времени на ненадежное приложение или даже атаки на систему с целью нарушить нормальное обслуживание пользователей. С другой стороны, мощное, надежное приложение может справедливо заслужить доступ ко всем типам системных ресурсов (например, файловая система, процесс создания, интерфейсы сети); вредоносное приложение может нанести серьезный ущерб этим ресурсам. Суть в том, что важные и иногда сложные проблемы безопасности должны решаться.

В некоторых ситуациях приемлемо просто попросить пользователя щелкать мышью по кнопке «ОК» при запросах. Язык Java предоставляет инструменты для применения любой политики безопасности, которую вы хотите. Однако от чего эта политика будет, в конце концов, зависеть, так это от уверенности в подлинности и целостности кода, о котором идет речь. Вот тут-то и выходят на сцену цифровые подписи.

Цифровые подписи совместно с сертификатами являются техниками проверки того, что данные в действительности исходят от источника, от которого они якобы исходят, и не были изменены по пути. Если компания подписывает свое чековое приложение, вы можете проверить, что приложение действительно идет из этого банка, а не от обманщика, и не было изменено. Соответственно, вы можете сделать так, чтобы ваш браузер доверял апплетам, которые имеют подпись компании.

История и перспективы развития Java

Когда столько всего происходит, тяжело уследить за тем, что доступно сейчас, что еще только обещается и что существует уже некоторое время. Следующие разделы составляют схематичный конспект, призванный упорядочить прошлое, настоящее и будущее Java.

Прошлое: Java 1.0 — Java 1.6

Язык Java 1.0 предоставлял базовую структуру разработки в среде Java: сам язык плюс пакеты, которые позволяют вам писать апплеты и простые приложения. Хотя Java 1.0 официально устарел, существует много апплетов, которые подчиняются правилам его API.

Язык Java 1.1 заменил 1.0, объединив главные улучшения в пакете абстрактно-оконного инструментария (Abstract Window Toolkit, AWT) (изначальная возможность Java GUI), нового шаблона событий и новых возможностей языка, таких как отражение, внутренние классы и многие другие важные свойства. Язык Java 1.1 исходно поддерживался большинством версий Netscape и Microsoft Internet Explorer много лет. По различным политическим причинам мир браузеров замер в этом состоянии на долгое время. Эта версия Java до сих пор считается базой для апплетов, хотя даже это изменится, если Microsoft прекратит поддержку Java на своих платформах.

Язык Java 1.2, названный компанией Sun «Java 2», стал крупным релизом в декабре 1988 года. Он содержал множество улучшений и добавлений, в основном в показателях набора интерфейсов API, которые были объединены в набор в стандартных распределителях. Самыми значительными добавлениями были включение пакета GUI Swing как центрального интерфейса API и нового полноценного API для рисования в 2D. Swing — это передовой набор инструментов Java для интерфейса пользователя со способностями, сильно превышающими старый AWT. (Swing, AWT и некоторые другие пакеты в разное время назывались JFC (Java Foundation Classes), или базовые классы Java.) Язык Java 1.2 также добавил подходящий интерфейс Collections API в Java. В язык Java 1.3, выпущенный в начале 2000-х, добавили лишь незначительные свойства, но изначально он был сфокусирован на производительности. С версией 1.3 язык Java стал значительно быстрее на многих платформах, и в инструментарии Swing было исправлено много ошибок. В это же время интерфейсы API предприятий Java, такие как Сервлеты и Enterprise JavaBeans, вполне развились.

Язык Java 1.4, выпущенный в 2002-м, интегрировал большой новый набор интерфейсов API и много долгожданных улучшений синтаксиса языка. Это включило языковые утверждения, регулярные выражения, интерфейсы API для предпочтений и регистрации, новую систему ввода/вывода для многоэтапных приложений, стандартную поддержку XML, фундаментальные улучшения в ATW и Swing и очень развитые интерфейсы API для сервлетов и веб-приложений Java.

Язык Java 5, выпущенный в 2004 году, был важным релизом, который ввел много долгожданных улучшений синтаксиса языка, включая параметры настройки, исчисления с типовой безопасностью, расширенные циклы с параметром, списки переменных аргументов, статические импорты, автоупаковку и распаковку примитивов, а также метаданные класса. Новый интерфейс API параллелизма предоставил мощные возможности потоков. Кроме того, были добавлены интерфейсы API для форматированной печати и парсинга, похожие на те, что есть в языке C. Интерфейс RMI (Remote Method Invocation, удаленный вызов метода) также был тщательно изучен, чтобы устранить потребность в компилируемых заглушках и фреймворках. Появились важные дополнения в стандартные интерфейсы XML API.

Язык Java 6, выпущенный в конце 2006-го, был довольно небольшим релизом, который не добавил никаких новых синтаксических свойств в язык Java, но упаковал интерфейсы API новых расширений, такие как для XML и веб-сервисов.

Java 7

Эта книга рассказывает обо всех последних и самых выдающихся усовершенствованиях последнего релиза Java 7. Этот релиз добавляет некоторые небольшие улучшения синтаксиса языка, такие как улучшение обработки исключений и управления ресурсами. Он также включает некоторые важные модификации API, такие как совсем новый интерфейс API файловой системы и дополнения к многим другим.

Это издание книги является первым после выпуска языка Java 5 и, следовательно, серьезно переработано, чтобы включить все изменения со времени релизов Java 6 и Java 7.

Ниже представлен короткий обзор самых важных свойств современной сердцевины интерфейса Java API.

Подключаемость базы данных Java (Java Database Connectivity /JDBC)

Общее средство взаимодействия с базами данных (появилось в Java 1.1).

Удаленный вызов метода (Remote Method Invocation /RMI)

Java является распределенной объектной системой. RMI позволяет вам вызывать методы на объекты, хранящиеся на сервере, работающем где-то в сети (появился в Java 1.1).

Безопасность Java

Средство контроля доступа к ресурсам системы, объединенное с унифицированным интерфейсом криптографии. Безопасность Java является основой для обозначенных классов, которые были описаны ранее.

Фундаментальные классы Java (JFC)

Обобщение для ряда свойств, включая компоненты интерфейса пользователя Swing; «встраиваемые вид и функции», что означает возможность интерфейса пользователя адаптироваться к внешнему виду платформы, которую вы используете, «перетаскивание» и доступность, что означает возможность интеграции со специальным программным обеспечением и аппаратным обеспечением для недееспособных людей.

Java 2D

Часть JFC; запускает высококачественную графику, манипуляции со шрифтами и рисование.

Интернационализация

Возможность писать программы, адаптирующиеся под язык, который пользователь хочет использовать; программа автоматически показывает текст на нужном языке (появилась в языке Java 1.1).

Интерфейс имен и папок Java (JNDI / Java Naming and Directory Interface)

Общий сервис для просмотра ресурсов. JNDI объединяет такие сервисы доступа к папкам, как LDAP, Novell's NDS и др.

Нижеследующее является интерфейсом API «стандартных расширений». Некоторые интерфейсы, такие как те, что работают с XML и веб-сервисами, связаны со стандартной редакцией Java; некоторые должны быть отдельно загружены и развернуты с вашим приложением на сервере.

JavaMail

Унифицированный интерфейс API для написания программного обеспечения для электронных писем.

Java 3D

Средство для разработки приложений с графикой 3D.

Java Media

Еще одно обобщение, которое включает Java 2D, Java 3D и медиафреймворк Java (фреймворк для координации показа многих разных типов медиа), Java Speech (для распознавания и синтеза речи), Java Sound (аудио высокого качества), Java TV (для интерактивного телевидения и подобных приложений) и др.

Сервлеты Java

Средство, которое позволяет вам писать серверные веб-приложения на Java.

Криптография Java

Действительное применение алгоритмов криптографии. (Этот пакет был отделен от безопасности Java по юридическим причинам.)

JavaHelp

Средство для написания систем помощи и объединения их в программы Java.

Enterprise JavaBeans

Компонентная архитектура для создания дистрибутивных приложений на стороне сервера.

Jini

Интересная технология распределенных компонентов, которая разработана для активации дистрибутивного вычисления, обнаружения и синхронизации расположения устройств от инструментов программного обеспечения до аппаратного обеспечения и бытовых электроприборов.

XML/XSL

Инструменты для создания и управления документами XML, проверки их, отображения в объектах Java и вне их, а также преобразования их при помощи таблицы стилей.

Веб-сервисы

Инструменты для создания и установки веб-сервисов на Java на основе простого протокола доступа к объектам.

В этой книге мы постараемся показать вам как можно больше свойств; к сожалению для нас (но к счастью для разработчиков программного обеспечения Java), среда Java стала такой богатой, что невозможно охватить все в одной книге.

Настоящее: Java 8

Java — живой и быстро развивающийся язык програмирования. В каждой новой версии продукта разработчики дополняют его удобными новыми инструментами, которые позволяют улучшить производительность программ.

Какие же новшества предлагает последняя версия Java 8? В языке появились лямбда-выражения, методы интерфейсов по умолчанию, ссылки на методы и повторяемые аннотации. Это основные новинки релиза.

Рассмотрим подробнее и на простых примерах каждый новый инструмент языка.

Методы интерфейсов по умолчанию

Теперь Java позволяет добавлять неабстрактные реализации методов в интерфейс, или так называемые методы расширения. Для этого используется ключевое слово `default`.

Рассмотрим пример такого добавления:

```
interface Formula {
double calculate(int a);
default double sqrt(int a) {
return Math.sqrt(a);
}
}
```

Помимо абстрактного метода `calculate` в интерфейсе `Formula` так же по умолчанию определяется метод `sqrt`. Классы, реализующие этот интерфейс, должны переопределить только абстрактный метод `calculate`. Метод по умолчанию `sqrt` будет доступен без переопределения.

```
Formula formula = new Formula() {
@Override
public double calculate(int a) {
```

```
return sqrt(a * 100);
}
};
formula.calculate(100); // 100.0
formula.sqrt(16); // 4.0
```

Здесь объект `formula` реализован как анонимный объект. Кроме того, код в этом примере достаточно избыточен: целых шесть строчек для такого простого вычисления, как `sqrt(a * 100)`. Немного позже мы рассмотрим более изящный способ реализации объектов с одним методом.

Лямбда-выражения

Прежде чем мы приступим к объяснению, что такое лямбда-выражения, давайте рассмотрим пример сортировки массива строк в предыдущих версиях языка. Код такой сортировки будет выглядит примерно так:

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
Collections.sort(names, new Comparator<String>() {
@Override
public int compare(String a, String b) {
return b.compareTo(a);
}
});
```

Здесь статический метод `Collections.sort` принимает список и компаратор, который используется для сортировки списка. Если вы уже практиковались в создании программ во время прочтения этого учебника или у вас имеется опыт программирования на Java, вам наверняка доводилось создавать анонимные компараторы для передачи их методу.

Java 8 предоставляет гораздо более короткий синтаксис — лямбда-выражения. Они добавлены в язык для того, чтобы вам не приходилось тратить время на создание анонимных объектов.

```
Collections.sort(names, (String a, String b) -> {
return b.compareTo(a);
});
```

Теперь код стал гораздо короче и более читаем. Давайте сделаем его еще короче и удобнее:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

Для однострочных методов позволяется опустить фигурные скобки и ключевое слово `return`. Вот так код стал еще короче:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Поскольку компилятору известны типы параметров, здесь их можно тоже опустить.

Рассмотрим несколько способов применения лямбда-выражений.

Функциональные интерфейсы

Как лямбда-выражения соответствуют системе типов языка Java? Каждой лямбде соответствует тип, представленный интерфейсом. Так называемый функциональный интерфейс должен содержать ровно один абстрактный метод. Каждое лямбда-выражение этого типа будет сопоставлено объявленному методу. Поскольку методы по умолчанию не являются абстрактными, вы можете добавлять в функциональный интерфейс сколько угодно таких методов.

Для лямбда-выражений разрешается использовать любые интерфейсы, содержащие только один абстрактный класс. Чтобы гарантировать, что написанный вами интерфейс соответствует этому требованию, следует использовать аннотацию `@FunctionalInterface`. Компилятор «поймет», что означает эта аннотация, и выдаст ошибку, если вас что-то отвлечет и вы добавите второй абстрактный метод в функциональный интерфейс.

Рассмотрим пример:

```
@FunctionalInterface
interface Converter<F, T> {
T convert(F from);
}
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted); // 123
```

Обратите внимание, что даже если убрать аннотацию `@Functional-Interface`, код в примере по-прежнему останется корректным.

Ссылки на методы и конструкторы

Предыдущий пример можно несколько упростить, если использовать статические ссылки на методы. Тогда он будет иметь следующий вид:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted); // 123
```

Java 8 позволяет вам передавать ссылки на методы или конструкторы. Для этого используется комбинация из двух символов двоеточия `::`.

В предыдущем примере продемонстрирована передача ссылки на статический метод.

Однако аналогичным образом можно ссылаться и на экземплярный метод.

Рассмотрим пример:

```
class Something {
String startsWith(String s) {
return String.valueOf(s.charAt(0));
}
}
Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted); // "J"
```

Теперь рассмотрим пример передачи ссылки на конструктор. Сначала создадим класс с несколькими конструкторами:

```
class Person {
String firstName;
String lastName;
Person() {}
Person(String firstName, String lastName) {
this.firstName = firstName;
this.lastName = lastName;
}
}
```

Теперь определим интерфейс для «фабрики», которая будет использоваться для создания новых «персон»:

```
interface PersonFactory<P extends Person> {
P create(String firstName, String lastName);
}
```

Теперь вместо реализации интерфейса соединим все вместе при помощи ссылки на конструктор:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

Здесь ссылка на конструктор создается с помощью `Person::new`. Компилятор автоматически выбирает подходящий конструктор, сигнатура которого совпадает с сигнатурой `PersonFactory.create`.

Области действия лямбд

Доступ к переменным внешней области действия из лямбда-выражения очень схож с доступом из анонимных объектов.

Вы можете ссылаться на переменные, объявленные как `final`, на экземплярные поля класса и статические переменные.

Рассмотрим пример:

```
int num = 1;
Converter<Integer, String> stringConverter = (from) -> String.valueOf(from
+ num);
stringConverter.convert(2); // 3
```

Однако в отличие от анонимных объектов переменная `num` не обязательно должна быть объявлена как `final`. Код, представленный в следующем примере, также будет работать:

```
int num = 1;
Converter<Integer, String> stringConverter = (from) -> String.valueOf(from
+ num);
stringConverter.convert(2); // 3
```

Однако переменная `num` не должна изменять значение. Следующий пример кода не будет скомпилирован:

```
int num = 1;
Converter<Integer, String> stringConverter = (from) -> String.valueOf(from
+ num);
num = 3;
```

Запись в переменную `num` в пределах лямбда-выражения также запрещена.

Доступ к полям и статическим переменным

В отличие от локальных переменных, мы можем записывать значения в экземплярные поля класса и статические переменные внутри лямбда-выражений. Это поведение хорошо знакомо по анонимным объектам.

Рассмотрим следующий пример:

```
class Lambda4 {
static int outerStaticNum;
int outerNum;
void testScopes() {
Converter<Integer, String> stringConverter1 = (from) -> {
outerNum = 23;
return String.valueOf(from);
};
Converter<Integer, String> stringConverter2 = (from) -> {
outerStaticNum = 72;
return String.valueOf(from);
};
}
}
```

Доступ к методам интерфейсов по умолчанию

Рассмотрим еще раз самый первый пример главы. В интерфейсе `Formula` определен метод по умолчанию `sqrt`, который доступен из каждой реализации интерфейса, включая анонимные объекты. Однако это не сработает в лямбда-выражениях.

Внутри лямбда-выражений запрещено обращаться к методам по умолчанию. Обратите внимание на приведенный в следующем примере код — этот код не будет скомпилирован.

```
Formula formula = (a) -> sqrt( a * 100);
```

Встроенные функциональные интерфейсы

JDK 1.8 содержит множество встроенных функциональных интерфейсов. Некоторые из них вам уже знакомы или даже хорошо известны по предыдущим версиям языка. Такие функциональные интерфейсы — это, например, `Comparator` или `Runnable`. Все они были поддержаны в лямбда-выражениях путем добавления аннотации `@FunctionalInterface`.

Помимо уже известных, в Java 8 появилось много новых функциональных интерфейсов, которые призваны облегчить жизнь разработчику. Некоторые интерфейсы хорошо известны по библиотеке Google Guava. Даже если вы незнакомы с этой библиотекой, вам обязательно стоит взглянуть, как эти интерфейсы дополнены некоторыми полезными методами расширений.

Предикаты

Предикаты — это функции, принимающие один аргумент и возвращающие значение логического типа `boolean`.

Интерфейс содержит различные методы по умолчанию, позволяющие строить сложные инструкции (например, «И», «ИЛИ», «НЕ»).

Рассмотрим пример:

```
Predicate<String> predicate = (s) -> s.length() > 0;
predicate.test("foo"); // true
predicate.negate().test("foo"); // false
Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Функции

Функции принимают один аргумент и возвращают некоторый результат. Методы по умолчанию могут использоваться для построения цепочек вызовов (`compose`, `andThen`).

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.
andThen(String::valueOf);
backToString.apply("123"); // "123"
```

Поставщики

Поставщики возвращают результат заданного типа. В отличие от функций, поставщики не принимают никаких аргументов. Пример:

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get(); // new Person
```

Потребители

Потребители представляют собой операции, которые производятся над одним переданным в потребитель аргументом.

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " +
p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

Компараторы

Компараторы описаны для предыдущих версий Java и хорошо известны. Java 8 добавляет в интерфейс различные методы по умолчанию. Рассмотрим пример:

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.
firstName);
Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");
comparator.compare(p1, p2); // > 0
comparator.reversed().compare(p1, p2); // < 0
```

Опциональные значения

Опциональные значения (`optionals`) не являются функциональными интерфейсами, однако являются удобным средством предотвращения ошибок типа `NullPointerException`. Это важная концепция, которая понадобится нам немного позже, поэтому давайте разберемся, как работают опциональные значения.

Опциональные значение — это, по сути, контейнер для значения, которое может быть равно `null`. Например, вам нужен метод, который обычно возвращает некоторое значение, однако иногда он должен возвращать пустое значение. Вместо того чтобы возвращать `null`, в Java 8 вы можете вернуть опциональное значение.

Рассмотрим пример кода:

```
Optional<String> optional = Optional.of("bam");
optional.isPresent(); // true
optional.get(); // "bam"
optional.orElse("fallback"); // "bam"
optional.ifPresent((s) -> System.out.println(s.charAt(0)));
// "b"
```

Потоки

Потоковый тип `java.util.Stream` представляет собой последовательность элементов, над которой можно производить различные операции. Операции над потоками могут быть промежуточными (`intermediate`) или конечными (`terminal`). Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же поток. Таким образом, вы можете строить цепочки из несколько операций над одним и тем же потоком. Потоки создаются на основе источников, например типов, реализующих коллекции `java.util.Collection`, такие как списки или множества (обратите внимание, что ассоциативные массивы здесь не поддерживаются). Операции над потоками могут выполняться как последовательно, так и параллельно.

Сначала давайте посмотрим, как работать с потоком последовательно. Для начала создадим источник в виде списка строк:

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

В Java 8 вы можете быстро создавать потоки, используя вызовы методов `Collection.stream()` или `Collection.parallelStream()`. Далее рассмотрим подробнее наиболее распространенные операции над потоками.

Фильтр

Операция фильтра (`Filter`) принимает предикат, который фильтрует все элементы потока. Эта операция является промежуточной, иными словами, она позволяет вызвать другую операцию (например, `forEach`), которая будет использоваться для обработки результата промежуточной операции. `ForEach` принимает функцию, которая вызывается для каждого элемента в уже отфильтрованном потоке. `ForEach` является конечной операцией. Она не возвращает никакого значения, поэтому дальнейший вызов потоковых операций для данного потока невозможен.

Рассмотрим пример кода:

```
stringCollection
.stream()
.filter((s) -> s.startsWith("a"))
.forEach(System.out::println);
// "aaa2", "aaa1"
```

Сортировка

Операция сортировки (`Sorted`) является промежуточной операцией, которая возвращает отсортированное представление потока. Элементы сортируются в обычном порядке, если не передается дополнительный компаратор:

```
stringCollection
.stream()
.sorted()
.filter((s) -> s.startsWith("a"))
.forEach(System.out::println);
// "aaa1", "aaa2"
```

Следует помнить, что `sorted` создает всего лишь отсортированное представление и не влияет на порядок элементов в исходной коллекции. Порядок строк в `stringCollection` остается нетронутым:

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Преобразование `map`

Операция преобразования `map` является промежуточной. Она преобразует каждый элемент в другой объект при помощи переданной функции. Следующий пример преобразует каждую строку в строку из символов в верхнем регистре. Однако вы также можете использовать `map` для преобразования каждого объекта в объект другого типа. Тип результирующего потока зависит от типа функции, которая передается при вызове `map`.

Рассмотрим пример:

```
stringCollection
.stream()
```

```
.map(String::toUpperCase)
.sorted((a, b) -> b.compareTo(a))
.forEach(System.out::println);
// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Соответствие

Для проверки, удовлетворяет ли поток заданному предикату, используются различные операции сопоставления (`match`). Все операции сопоставления являются конечными и возвращают результат логического типа `boolean`, который может иметь значение `true` или `false`, соответственно 1 или 0.

Пример:

```
boolean anyStartsWithA =
stringCollection
.stream()
.anyMatch((s) -> s.startsWith("a"));
System.out.println(anyStartsWithA); // true
boolean allStartsWithA =
stringCollection
.stream()
.allMatch((s) -> s.startsWith("a"));
System.out.println(allStartsWithA); // false
boolean noneStartsWithZ =
stringCollection
.stream()
.noneMatch((s) -> s.startsWith("z"));
System.out.println(noneStartsWithZ); // true
```

Количество элементов

Операция `Count` является конечной и возвращает количество элементов в потоке. Возвращаемое значение имеет числовой тип `long`.

```
long startsWithB =
stringCollection
.stream()
.filter((s) -> s.startsWith("b"))
.count();
System.out.println(startsWithB); // 3
```

Сокращение

Эта операция производит свертку элементов потока по заданной функции, она является конечной. Результат выполнения функции — опциональное значение.

Рассмотрим пример:

```
Optional<String> reduced =
stringCollection
.stream()
.sorted()
.reduce((s1, s2) -> s1 + "#" + s2);
reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

Параллельные потоки

Как говорилось ранее, потоки могут быть последовательными и параллельными. Операции над последовательными потоками выполняются в одном потоке процессора, операции над параллельными потоками используют несколько потоков процессора.

Следующий пример демонстрирует, как можно легко увеличить скорость работы, используя параллельные потоки.

Для начала создадим большой список из уникальных элементов:

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
UUID uuid = UUID.randomUUID();
values.add(uuid.toString());
}
```

Теперь измерим время сортировки этого списка используя различные методы сортировки.

Последовательная сортировка

```
long t0 = System.nanoTime();
long count = values.stream().sorted().count();
System.out.println(count);
long t1 = System.nanoTime();
```

```
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0); System.out.
println(String.format("sequential sort took: %d ms", millis));
// sequential sort took: 899 ms
```

Время сортировки последовательным способом заняло 899 мс.

Параллельная сортировка

```
long t0 = System.nanoTime();
long count = values.parallelStream().sorted().count();
System.out.println(count);
long t1 = System.nanoTime();
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0); System.out.
println(String.format("parallel sort took: %d ms", millis));
// parallel sort took: 472 ms
```

Время сортировки параллельным способом заняло 472 мс.

Очевидно, что оба примера кода практически идентичны, однако параллельная сортировка почти в два раза быстрее. Все, что нужно сделать, это заменить вызов `stream()` на `parallelStream()`.

Ассоциативные массивы

Ранее уже говорилось, что ассоциативные массивы (`maps`) не поддерживают потоки. Вместо этого ассоциативные массивы в новом релизе Java поддерживают различные полезные методы, которые решают часто встречающиеся при разработке приложений задачи.

```
Map<Integer, String> map = new HashMap<>();
for (int i = 0; i < 10; i++) {
map.putIfAbsent(i, "val" + i);
}
map.forEach((id, val) -> System.out.println(val));
```

Давайте добавим несколько комментариев к приведенному выше примеру. Итак, `putIfAbsent` позволяет нам не писать дополнительные проверки на `null`, `forEach` принимает потребителя, который производит операцию над каждым элементом массива. Следующий пример демонстрирует, как код используется для вычислений при помощи различных функций:

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3); // val33
```

```
map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9); // false
map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23); // true
map.computeIfAbsent(3, num -> "bam");
map.get(3); // val33
```

Чтобы удалить объект по ключу, в случае когда объект ассоциирован с ключом, можно использовать следующий код:

```
map.remove(3, "val3");
map.get(3); // val33
map.remove(3, "val33");
map.get(3); // null
```

И еще один полезный метод:

```
map.getOrDefault(42, "not found"); // not found
```

Если нужно объединить записи двух массивов, используйте следующий пример:

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9); // val9
map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9); // val9concat
```

В случае отсутствия ключа функция `Merge` создает новую пару «ключ/значение». В противном случае — вызывает функцию объединения для существующего значения.

API для работы с датами

Java 8 содержит совершенно новый интерфейс программирования приложений (API) для работы с датами и временем, расположенный в пакете `java.time`. Рассмотрим подробно ряд наиболее важных разделов нового интерфейса программирования приложений.

Тип данных `clock`

Тип `Clock` предоставляет доступ к текущей дате и времени. В нем хранится информация о часовых поясах, и его можно использовать вместо вызова функции `System.currentTimeMillis()` для возвращения мил-

лисекунд. Такая точная дата также может быть представлена классом `Instant`. Объекты этого класса могут быть использованы для создания объектов более старого типа `java.util.Date`.

Рассмотрим примеры использования типа `Clock`:

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();
Instant instant = clock.instant();
Date legacyDate = Date.from(instant);
// legacy java.util.Date
```

Часовые пояса

Часовые пояса в Java 8 представлены типом `ZoneId`. Доступ к ним можно получить при помощи статических методов. Часовые пояса содержат значения смещений, которые важны для конвертации даты и времени в соответствующее местное время.

```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());
// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

Тип `LocalTime`

Тип `LocalTime` содержит в себе время с учетом часового пояса. Например, 10pm или 17:30:15. В следующем примере образуются два местных времени для часовых поясов, определенных выше. Затем оба времени сравниваются, и вычисляется разница между ними в часах и минутах.

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);
System.out.println(now1.isBefore(now2)); // false long
hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2); System.out.
println(hoursBetween); // -3
System.out.println(minutesBetween); // -239
```

Тип `LocalTime` содержит различные встроенные методы, которые упрощают создание новых экземпляров объектов, а также парсинг строковых констант. Для примера рассмотрим еще небольшой блок кода:

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late); // 23:59:59
DateTimeFormatter germanFormatter =
DateTimeFormatter
.ofLocalizedTime(FormatStyle.SHORT)
.withLocale(Locale.GERMAN);
LocalTime leetTime = LocalTime.parse("13:37", germanFormatter); System.
out.println(leetTime); // 13:37
```

Представление локальной даты `LocalDate`

Тип `LocalDate` представляет конкретную дату, например, 2014-03-11. Объекты `LocalDate` неизменяемы и являются аналогом объектов типа `LocalTime`. Рассмотрим пример, демонстрирующий вычисление новой даты путем сложения или вычитания дней, месяцев или лет. При этом необходимо помнить, что каждая операция возвращает новый экземпляр.

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);
LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek); // FRIDAY
```

Также есть возможность создавать новые объекты экземпляра `LocalDate` с помощью парсинга строк. Следующий код является примером создания такого экземпляра:

```
DateTimeFormatter germanFormatter =
DateTimeFormatter
.ofLocalizedDate(FormatStyle.MEDIUM)
.withLocale(Locale.GERMAN);
LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter); System.
out.println(xmas); // 2014-12-24
```

Представление локальной даты и времени `LocalDateTime`

Тип `LocalDateTime` представляет собой комбинацию даты и времени. Объекты `LocalDateTime` неизменяемы и работают аналогично объектам

типов LocalTime и LocalDate. Для извлечения конкретных значений из объекта даты-времени имеется ряд функций:

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23,
59, 59);
DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek); // WEDNESDAY
Month month = sylvester.getMonth();
System.out.println(month); // DECEMBER
long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY); System.
out.println(minuteOfDay); // 1439
```

С помощью добавления информации о часовом поясе можно получить Instant.

```
Instant instant = sylvester
.atZone(ZoneId.systemDefault())
.toInstant();
Date legacyDate = Date.from(instant);
System.out.println(legacyDate); // Wed Dec 31 23:59:59 CET 2014
```

Форматирование объекта даты-времени работает так же, как и форматирование объекта даты или объекта времени. Для этого можно использовать библиотечные или написанные собственноручно шаблоны.

```
DateTimeFormatter formatter =
DateTimeFormatter
.ofPattern("MMM dd, yyyy - HH:mm");
LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13",
formatter);
String string = formatter.format(parsed);
System.out.println(string); // Nov 03, 2014 - 07:13
```

В отличие от типа java.text.NumberFormat, новый тип DateTimeFormatter является неизменяемым и потокобезопасным.

Аннотации

Аннотации в Java 8 являются повторяемыми. Чтобы сразу разобраться, что это такое, давайте рассмотрим пример.

Для начала определим аннотацию-обертку, которая содержит массив аннотаций:

```
@interface Hints {
Hint[] value();
}
@Repeatable(Hints.class)
@interface Hint {
String value();
}
```

Java 8 позволяет нам использовать множество аннотаций одного типа путем указания аннотации как `@Repeatable`. Рассмотрим два варианта использования аннотаций.

Вариант 1: использовать аннотацию-контейнер (старый способ)

```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

Вариант 2: использовать повторяемую аннотацию (новый способ)

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

При использовании второго варианта компилятор автоматически подставляет аннотацию `@Hints`. Это важно при чтении информации об аннотациях через рефлексию.

```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint); // null
Hints hints1 = Person.class.getAnnotation(Hints.class); System.out.
println(hints1.value().length); // 2
Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class); System.out.
println(hints2.length); // 2
```

Хотя аннотация `@Hints` не объявлялась в классе `Person`, тем не менее она доступна нам при вызове `getAnnotation(Hints.class)`. Однако более удобным является метод `getAnnotationsByType`, который напрямую предоставляет доступ ко всем аннотациям `@Hint`.

Кроме этого, аннотации в Java 8 можно использовать еще на двух элементах:

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

На этом перечень основных нововведений в Java 8 окончен. Более близко познакомиться и детально изучить все тонкости релиза вы можете в процессе создания своих приложений и проектов.

Будущее

Изменения в Java стали менее частыми, поскольку язык Java возмужал с течением времени, но Java продолжает быть одной из самых популярных платформ для разработки приложений. Это особенно справедливо по отношению к веб-сервисам, фреймворкам веб-приложений и инструментам XML. Хотя Java и не доминирует на мобильных платформах, для чего, казалось бы, он предназначен, язык Java и центральные интерфейсы API применяются в программировании мобильных операционных систем для Google Android, которые используются на миллиардах устройств по всему миру. В лагере Microsoft язык C#, образованный от Java, унаследовал многое от разработки на .NET и перенес синтаксис и шаблоны ядра Java на эти платформы.

Вероятно, самые захватывающие изменения Java сегодня тяготеют к легковесности, более простым фреймворкам для бизнеса и интеграции платформы Java с динамическими языками для написания веб-страниц и расширений. Еще более интересная работа впереди.

Доступность

У вас есть несколько вариантов разработки сред и систем выполнения Java. Набор для разработки Java-приложений Oracle (JDK, Java Development Kit) доступен для Mac OS X, Windows и Linux.

Посетите вебсайт Java Oracle, чтобы узнать больше о том, как получить последний набор инструментов JDK.

Существует также целое множество популярных интегрированных сред разработки Java. В этой книге мы рассмотрим две из них: Eclipse от IBM и NetBeans IDE от Oracle.

Эти цельные среды разработки позволят вам писать, тестировать и упаковывать программное обеспечение, имея передовые инструменты под вашими пальцами. В то время как Eclipse, без сомнений, является самой популярной средой и имеет открытый код, автор предпочитает IDE Intellij IDEA компании JetBrains, которая сейчас также имеет бесплатное издание сообщества.

Глава 2
ПЕРВОЕ ПРИМЕНЕНИЕ

Перед тем как погрузиться в обсуждение языка Java, давайте помочим ноги, погрузившись в некий рабочий код, и поплещемся там. В этой главе мы построим дружелюбное маленькое приложение, которое иллюстрирует многие концепции, использованные в книге. Мы используем эту возможность, чтобы представить общие свойства языка и приложений Java.

Эта глава также служит кратким вступлением в объектно-ориентированные и многопоточные аспекты Java. Если эти идеи новы для вас, мы надеемся, что первое столкновение с ними будет простым и приятным опытом. Если вы работали с другой объектно-ориентированной или многопоточной средой, вы тем более должны оценить простоту и элегантность Java. Эта глава предназначена для того, чтобы показать вам язык Java с высоты птичьего полета и дать почувствовать, как он используется. Если у вас возникнут проблемы с каким-либо понятием, представленным здесь, не сомневайтесь, что оно будет рассмотрено подробно далее в книге. Сложно переоценить важность собственных экспериментов во время изучения новых понятий здесь и во всей книге. Не просто читайте примеры — воспроизводите их. Исходный код этих примеров и всех примеров в этой книге вы можете найти на нашем сайте. Компилируйте программы и пробуйте их. Затем превращайте наши примеры в ваши примеры: играйте с ними, меняйте их поведение, ломайте их, чините их и, будем надеяться, вы весело проведете за ними время.

Инструменты и среда Java

Хотя можно писать, компилировать и запускать приложения Java только при помощи *набора для разработки Java-приложений* (JDK) и простого текстового редактора (например, vi, Блокнот (Notepad) и др.), сегодня огромная часть Java-кода пишется благодаря *интегрированной среде разработки* (IDE). Преимущества использования

IDE включают целостный взгляд на исходный Java-код с подсветкой синтаксиса, навигационной помощью, контролем за исходным кодом, интегрированной документацией, расстановкой, реструктуризацией и размещением — все у вас под рукой. Таким образом, мы пропустим академическую консольную трактовку и начнем с самой популярной среды IDE — Eclipse. Если вы враждебно относитесь к использованию IDE, можете со спокойной душой использовать консольные инструкции `javac HelloJava.java` для компиляции и `java Hello Java` для запуска последующих примеров.

Компания IBM изначально была инициатором The Eclipse Project в 2001 году, что подвигло консорциум поставщиков программного обеспечения создать расширяемую среду для разработки, чтобы конкурировать с популярной в ту пору средой Visual Studio от Microsoft. Сегодня среда Eclipse превратилась в мощную платформу с открытым кодом, поддерживаемую как отдельными людьми, так и корпорациями, финансируемую процветающей экосистемой плагинов и фреймворков. Хотя Java является самым популярным языком, ассоциирующимся с Eclipse, IDE поддерживает дюжину языков. В этой главе мы немного познакомимся с Eclipse. Для более исчерпывающей информации смотрите приложение A.

Eclipse требует установки *среды выполнения Java* (JRE, Java Runtime Environment). Эта глава освещает свойства языка Java 7, так что, хотя примеры этой главы и работают с более старыми версиями, лучше установить JDK 7, чтобы убедиться, что все примеры в этой книге компилируются. JDK включает JRE, как и инструменты разработки. Вы можете проверить, какая версия у вас установлена, введя java-version в командной строке. Если Java нет или если ее версия более древняя, чем JDK 7 (также ошибочно называемая JDK 1.7), вам нужно будет скачать самую последнюю версию со страницы загрузки Oracle. Все, что необходимо для работы с примерами в этой книге, — это базовый JDK, который является первой опцией в верхнем левом углу страницы загрузки.

Eclipse является IDE с открытым кодом, которую можно скачать на Eclipse.org. Для задач этой книги и общего знакомства с Java достаточно загрузки Eclipse Classic. Убедитесь, что архитектура вашего JDK и Eclipse совпадает. Иными словами, не используйте 64-битный JDK и 32-битную версию Eclipse или наоборот. Загрузка является сжатым архивом *.zip* для Windows и *.tar.gz* для OS X и Linux. Дважды щелкните мышью, чтобы распаковать и запустить установщик.

Конфигурация Eclipse и создание проекта

При первом использовании Eclipse вам необходимо будет выбрать рабочее пространство. Это корневая папка для хранения нового проекта, который вы создадите в Eclipse. Местоположение по умолчанию находится внутри самой папки приложения, и это, скорее всего, не то, что вам нужно. Выберите местоположение и нажмите кнопку **ОК**.

Eclipse встречает вас экраном приветствия. Закройте это окно, закрыв вкладку приветствия в приложении. Если вы захотите вернуться позже и пройти руководство Eclipse и родственные разделы справки, вы можете вызвать это окно, выбрав команду меню **Help ⇒ Welcome** (Справка ⇒ Окно приветствия).

Но прежде, чем мы продолжим: Eclipse хранит всю информацию о конфигурации в папке *configuration* в папке установки Eclipse. Если в любой момент этого вступления вы почувствуете, что что-то не так, и захотите начать с самого начала, вы можете покинуть приложение и удалить эту папку.

Вы также, возможно, захотите удалить объекты вашего рабочего пространства, поскольку они содержат состояние каждого проекта. Если вы захотите сбросить все окна приложения до их установок по умолчанию менее радикально, вы можете выбрать команду меню **Window ⇒ Reset Perspective** (Окно ⇒ Сброс Перспектива). Мы еще поговорим об этой перспективе позже.

Мы создадим проект, который будет содержать все наши примеры. Выберите команду меню **File ⇒ New ⇒ Java Project** (Файл ⇒ Новый ⇒ Проект Java) из меню приложения и введите текст **Learning Java** в поле **Project Name** (Название проекта) в верхней части диалогового окна, как показано на рис. 2.1.

Убедитесь, что в группе элементов управления **JRE** выбрана версия среды JavaSE-1.7, как показано на рисунке, и нажмите кнопку **Next** (Далее) в нижней части диалогового окна.

Затем вам нужно будет установить путь сборки в системную библиотеку Java 7. Выберите вкладку **Libraries** (Библиотеки) и удалите библиотеку Java 1.6. Нажмите кнопку **Add Library** (Добавить библиотеку) и выберите библиотеку JavaSE-1.7. Сейчас среда Eclipse сконфигурирована на использование Java 7. Нажмите кнопку **Finish** (Готово).

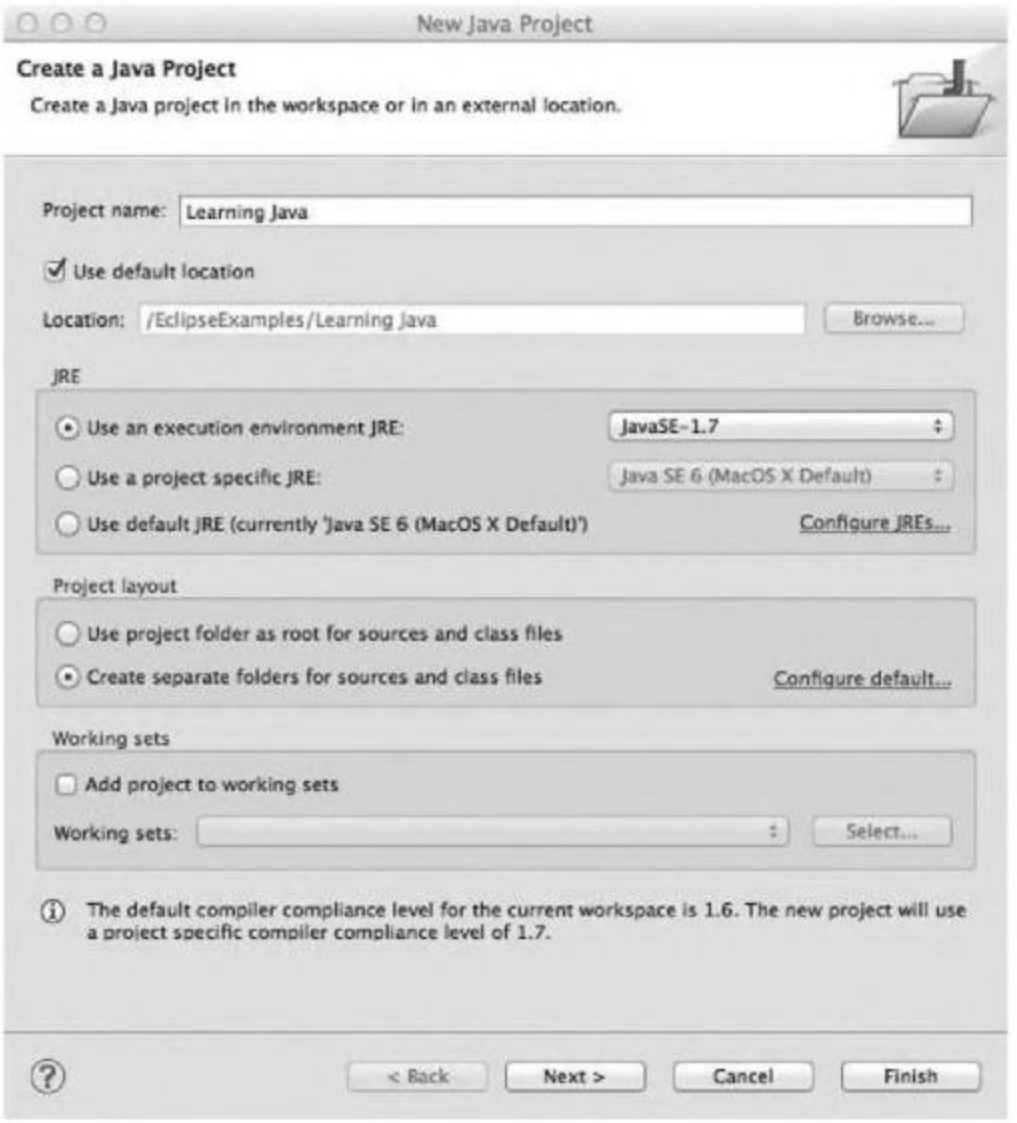

Рис. 2.1. Диалоговое окно для создания нового проекта Java

Импорт примеров по изучению Java

Давайте загрузим примеры из этой книги. Вы можете найти zip-файл, содержащий все примеры из этой книги, по адресу **eksmo.ru/smv/Primers.zip**. Версия примеров Eclipse содержится в архиве с именем *examples-eclipse.zip*. (Файл *examples.zip* содержит те же самые примеры, но упакованные немного по-другому, без файлов проекта Eclipse.)

Далее мы импортируем zip-файл с примерами. Выберите команду **File ⇒ Import** (Файл ⇒ Импорт), чтобы открыть мастер импорта. Выберите команду **Archive File** (Архивный файл) в качестве источника и нажмите кнопку **Next** (Далее). Смотрите рис. 2.2.

Нажмите кнопку **Browse** (Обзор) и выберите файл *examples-eclipse.zip*, как показано на рис. 2.3. Нажмите кнопку **Overwrite existing resources without warning** (Перезаписать существующие ресурсы без уведомления), так что наш проект на Eclipse перезапишет пустой проект в вашем новом. Нажмите кнопку **Finish** (Готово).

Теперь Eclipse импортирует все файлы из архива и немедленно начнет создавать источник в фоновом режиме (небольшой индикатор выполнения внизу экрана покажет это). Слева находится Проводник

Пакетов. Он показывает древовидное меню пакетов Java, библиотек и ресурсов нашего проекта. Нажмите на ручки папки, чтобы раскрыть дерево и увидеть исходные папки для каждой главы книги. Теперь мы готовы начать программирование!



Рис. 2.2. Диалоговое окно проекта Java

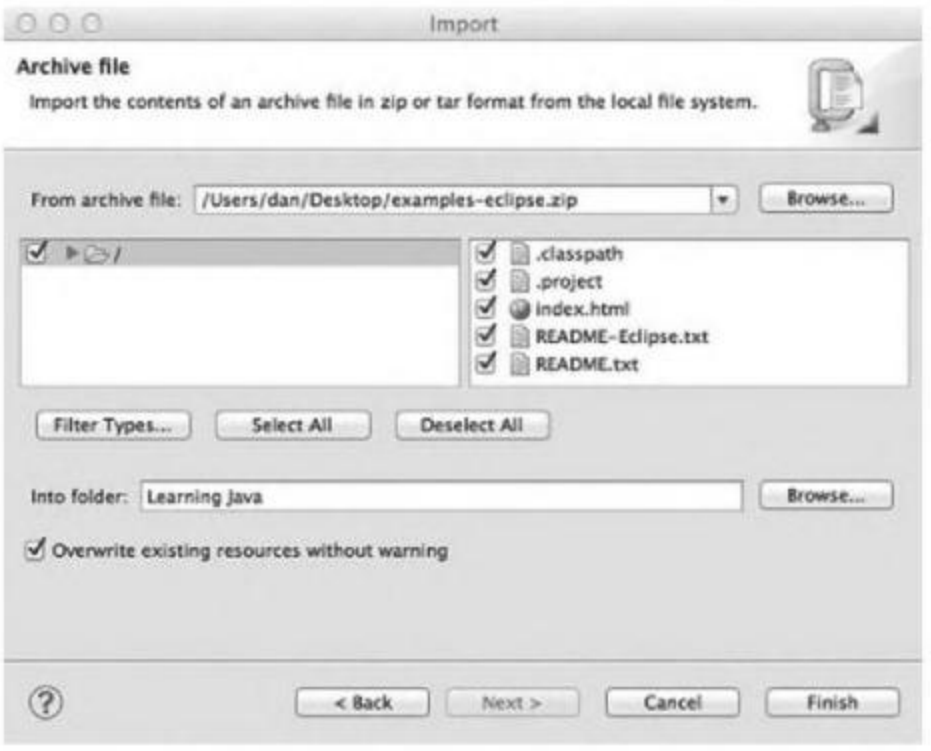

Рис. 2.3. Диалоговое окно нового проекта Java

HelloJava

В традиции вступительных программных текстов, мы начнем с эквивалента приложению Hello World от Java — HelloJava.

Мы совершим четыре прохода по этому примеру перед тем, как мы закончим (HelloJava, HelloJava2, др.), добавляя свойства и вводя новые понятия. Но давайте начнем с минималистской версии:

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

Эта программа из пяти строк объявляет класс, называющийся `HelloJava` и метод `main()`. Она использует предопределенный метод, называемый `println()`, чтобы вывести какой-то текст. Это *консольная программа*, что означает, что она работает в командном окне или окне DOS и выводит свой результат там. Это немного старомодно на наш вкус, поэтому прежде чем продолжить, мы реализуем в классе `HelloJava` *графический интерфейс пользователя* (GUI). Пока не беспокойтесь о коде; просто выполняйте последовательность, и мы вернемся к объяснению вскоре.

Вместо строки, содержащей метод `println()`, мы используем объект `JFrame`, чтобы вывести окно на экран. Мы можем начать с замены строки `println` тремя следующими строками:

```
JFrame frame = new JFrame( "Hello, Java!" );
frame.setSize( 300, 300 );
frame.setVisible( true );
```

Этот фрагмент создает объект `JFrame` с заголовком Hello, Java!. `JFrame` является графическим окном. Чтобы его показать, мы просто конфигурируем его размер на экране, используя метод `setSize()`, и делаем его видимым, вызывая метод `setVisible()`.

Если бы мы остановились на этом, мы бы увидели пустое окно на экране с нашим баннером Hello, Java! в качестве заголовка. Мы бы хотели, чтобы наше сообщение было внутри окна, а не просто небрежно написано в его верхней части. Чтобы поместить что-то в окно, нам понадобится еще пара строк. Следующий полный пример добавляет объект `JLabel`, чтобы показать текст по центру нашего окна. Дополнительная строка импорта вверху необходима, чтобы сказать Java, где найти классы `JFrame` и `JLabel` (определения объектов `JFrame` и `JLabel`, которые мы используем).

```
import javax.swing.*;
public class HelloJava {
```

```
public static void main( String[] args ) {
JFrame frame = new JFrame( "Hello, Java!" );
JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
frame.add(label);
frame.setSize( 300, 300 );
frame.setVisible( true );
}
}
```

Теперь, чтобы скомпилировать и запустить этот исходник, выберите класс `ch02/HelloJava.java` из проводника пакетов слева и нажмите кнопку **Run** (Выполнить) в панели инструментов сверху. Кнопка **Run** (Выполнить) представлена круглым зеленым кругом с белой стрелкой, показывающей вправо. См. рис. 2.4.

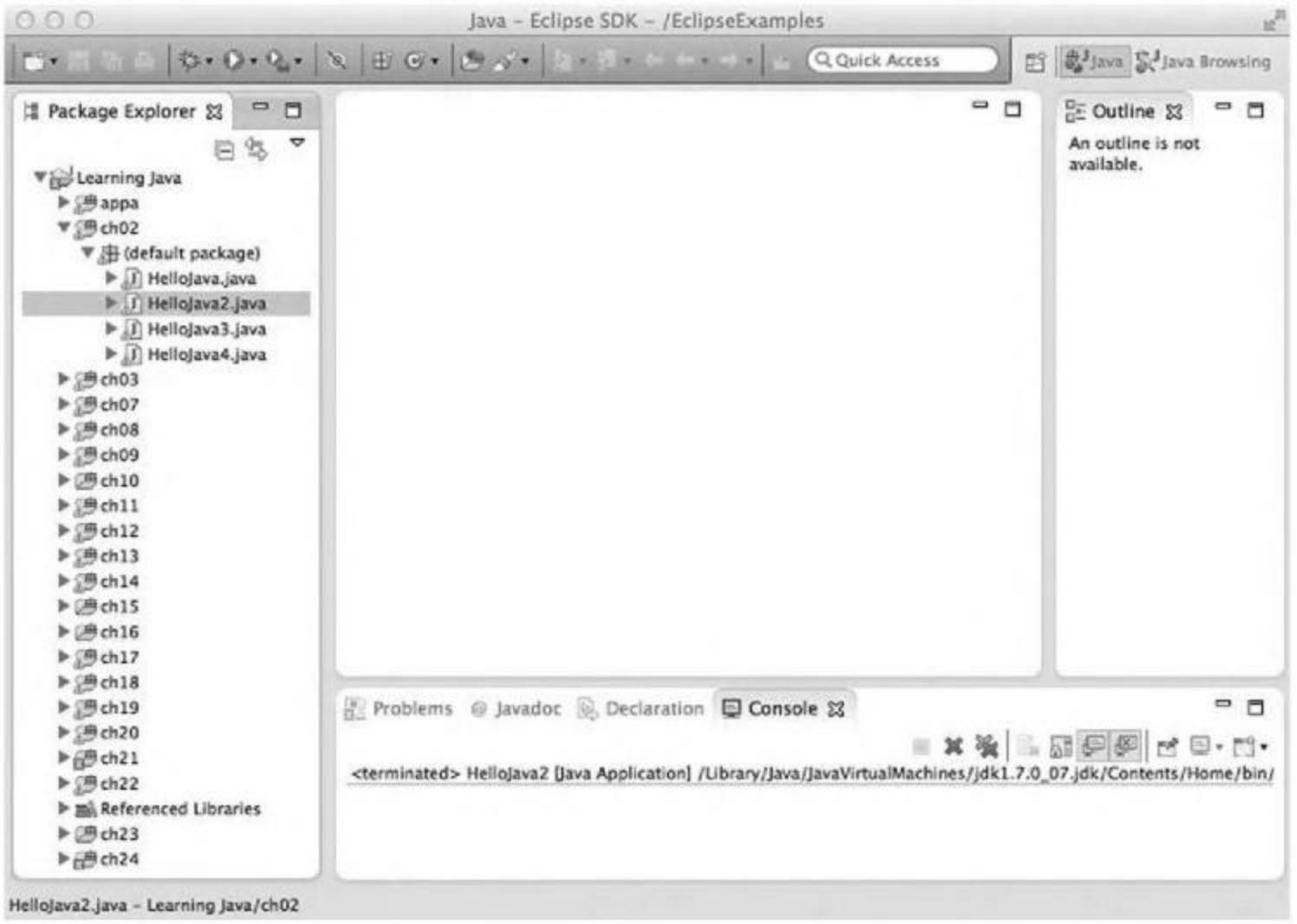

Рис. 2.4. Запуск приложения `HelloJava`

Вы должны увидеть объявление, показанное на рис. 2.5. Поздравляем, вы запустили свое первое Java-приложение! Уделите время, чтобы насладиться сиянием вашего монитора.

Обратите внимание на то, что когда вы нажмете на элемент закрытия окна, оно пропадет, но программа все еще выполняется. (Мы исправим это поведение при закрытии в одной из следующих версий примера.)

Чтобы остановить Java-приложение в Eclipse, нажмите большую красную кнопку в консольном окне. Если вы запускаете пример в командной строке, нажмите сочетание клавиш **Ctrl+C**. Обратите внимание, что ничего не мешает вам запускать больше одного образца (копии) приложения за раз.



Рис. 2.5. Результат работы приложения `HelloJava`

Возможно, `HelloJava` и маленькая программа, но много чего происходит за кулисами. Эти несколько строк представляют верхушку айсберга. То, что лежит под поверхностью, является слоями функциональности, предоставленной языком Java и его библиотеками базовых классов. Запомните, что в этой главе мы собираемся пройти большое расстояние быстро в попытке показать вам полную картину. Мы попытаемся дать достаточно деталей для хорошего понимания того, что происходит в каждом примере, но мы отложим подробные объяснения до нужных глав. Это касается элементов языка Java и объектно-ориентированных понятий, которые мы к ним применяем. Принимая во внимание все вышесказанное, давайте взглянем на то, что происходит в нашем первом примере.

Классы

Первый пример определяет класс, который называется `HelloJava`.

```
public class HelloJava {
...
```

Классы — фундаментальные строительные блоки многих объектно-ориентированных языков. *Класс* является группой элементов данных со связанными функциями, которые могут выполнять операции над этими данными. Элементы данных в классе называются *переменными*, иногда *полями*; в Java функции называются *методами*. Первичные преимущества объектно-ориентированного языка — это связи между данными и функционалом в элементах класса, а также возможность классов *ин-*

капсулировать или скрывать детали, освобождая разработчика от беспокойства о деталях низкого уровня.

В приложении класс может представлять что-то конкретное, например кнопку на экране или информацию в электронной таблице, или это может быть что-то более абстрактное, например алгоритм сортировки или, возможно, эмоция грусти персонажа видеоигры. Класс, представляющий таблицу, может, например, представлять значения ее индивидуальных ячеек или методов, которые производят действия над этими ячейками, например: «очистить ряд» или «вычислить значения».

Наш класс `HelloJava` является целым Java-приложением в одном классе. Он определяет только один метод `main()`, который содержит тело нашей программы:

```
public class HelloJava {
public static void main( String[] args ) {
...
```

когда запускается приложение, первым вызывается этот метод `main()`. Кусок, помеченный как `String [] args`, позволяет нам передавать *консольные аргументы* приложению. Мы рассмотрим метод `main()` в следующем разделе. В конце концов, мы обнаружим, что хотя эта версия класса `HelloJava` не определяет никакие переменные, как часть его класса, она использует две переменных, `frame` и `label`, внутри метода `main()`. Вскоре мы подробнее поговорим о переменных.

Метод `main()`

Как мы видели, запуская наш пример, запуск приложения Java означает подбор определенного класса и передачу его имени в качестве аргумента виртуальной машине Java. Когда мы это делали, команда `java` «смотрела» в наш класс `HelloJava`, чтобы увидеть, содержит ли он специальный метод с именем `main()` правильной формы. Он содержал, и поэтому приложение выполнялось. Если бы его там не было, мы бы получили сообщение об ошибке. Метод `main()` является точкой входа приложения. Каждое автономное Java-приложение включает хотя бы один класс с методом `main()`, который производит необходимые действия, чтобы начать остальную часть программы.

Наш метод `main()` вызывает окно `(JFrame)`, чтоб хранить визуальный выход класса `HelloJava`. Сейчас он выполняет всю работу в приложе-

нии. Но в объектно-ориентированном приложении мы обычно делегируем полномочия многим другим классам. В следующем воплощении нашего примера мы выполним такое расщепление — создадим второй класс — и увидим, что в то время, как пример последовательно эволюционирует, метод `main()` остается более-менее тем же, просто храня процесс запуска.

Давайте быстро пробежимся по нашему методу `main()`, просто чтобы знать, что он делает. Во-первых, `main()` создает `JFrame` — окно, которое будет хранить наш пример:

```
JFrame frame = new JFrame("Hello, Java!");
```

Слово `new` в этой строчке кода очень важно. `JFrame` — это имя класса, который представляет окно на экране, но сам класс является просто шаблоном, как план здания. Ключевое слово `new` говорит Java о распределении памяти и в действительности создает отдельный объект `JFrame`. В этом случае аргумент в круглых скобках говорит `JFrame`, что показывать в строке заголовка. Мы могли бы пропустить текст Hello, Java и использовать пустые круглые скобки, чтобы создать объект `JFrame` без заголовка, но только потому, что `JFrame` специально позволяет сделать нам это.

Когда обрамляющие окна только созданы, они очень маленькие. Перед тем как мы покажем `JFrame`, установим более значительный размер:

```
frame.setSize( 300, 300 );
```

Это пример вызова метода к определенному объекту. В этом случае установленный метод `setSize()` определен классом `JFrame` и влияет на отдельный объект `JFrame`, который мы поместили в переменную `frame`. Подобно фрейму мы также создаем экземпляр `JLabel`, чтобы хранить наш текст в окне:

```
JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
```

Объект `JLabel` очень похож на физическую метку. Он удерживает текст в отдельной позиции — в данном случае в нашем фрейме. Это очень объектно-ориентированная концепция: использование объекта для содержания некоторого текста вместо простого вызова метода для «отрисовки» текста и продолжения. Рациональное зерно этого будет яснее позже.

Далее мы должны поместить ярлык во фрейм, который мы создали:

```
frame.add( label );
```

Здесь мы вызываем метод, который называется `add()`, чтобы поместить наш ярлык в `JFrame`. `JFrame` является подобием контейнера для хранения. Мы поговорим об этом позже. Финальной задачей `main()` является показ фрейма и его содержимого, которое в другом случае будет невидимым. Невидимое окно представляет собой довольно скучное приложение.

```
frame.setVisible( true );
```

В этом весь метод `main()`. Во время работы с этими примерами в этой главе он будет в основном оставаться неизменным, в то время как станет эволюционировать класс `HelloJava`.

Классы и объекты

Класс является макетом части приложения; он хранит методы и переменные, которые формируют этот компонент. Многие индивидуальные рабочие копии данного класса могут существовать, пока приложение активно. Эти индивидуальные воплощения называются *экземплярами* класса или *объектами*. Два экземпляра данного класса могут содержать разные данные, но они всегда имеют одни и те же методы.

Как пример рассмотрите класс `Button`. Существует только один класс `Button`, но приложение может создать много разных объектов `Button`, каждый из которых будет экземпляром того же класса. Более того, два экземпляра `Button` могут содержать разные данные, возможно, иметь разный внешний вид и производить различные действия. В этом смысле класс может считаться формой для создания объекта, который он представляет, чем-то наподобие формочки для печенья, штампующей рабочие экземпляры самой себя в памяти компьютера. Как вы увидите позже, в этом есть что-то большее — класс на самом деле может делить информацию между экземплярами, но этой информации пока будет достаточно. В главе 5 содержится вся информация о классах и объектах.

Термин *объект* является очень общим и в некоторых других ситуациях используется почти поочередно с *классом*. Объекты являются абстрактными сущностями, к которым обращаются объектно-ориен-

тированные языки в одной или другой форме. Мы будем использовать *объект* как общее обозначение для представителя класса. Мы можем, таким образом, ссылаться на экземпляр класса `Button` как на кнопку, объект `Button` или, без разбора, как на объект.

Метод `main()` в предыдущем примере создает один экземпляр класса `JLabel` и показывает его как экземпляр класса `JFrame`. Можно изменять `main()`, чтобы создавать много экземпляров `JLabel`, в том числе каждый в отдельном окне.

Переменные и типы классов

В Java каждый класс определяет новый *тип* (тип данных). Переменная может быть объявлена как представитель этого типа и затем содержать представителей этого класса. Переменная может быть, например, типа `Button` и содержать экземпляр класса `Button` или типа `SpreadSheetCell` и содержать объект `SpreadSheetCell`, так же как она может быть представителем любого более простого типа, например `int` или `float`, которые представляют числа. Тот факт, что переменные имеют типы и не могут просто содержать любой тип объектов, является еще одним свойством языка, которое обеспечивает безопасность и правильность кода. Не учитывая переменные, используемые в методе `main()` в данный момент, только одна переменная объявляется в нашем простом примере `HelloJava`. Она находится в объявлении самого метода:

```
public static void main( String [] args ) {
```

Как функции в других языках, метод в Java объявляет список переменных, которые он принимает за *аргументы* или *параметры*, и определяет типы этих переменных. В этом случае метод `main` требует, чтобы при его вызове ему передавался список объектов `String` в переменную, именуемую `args`. `String` является фундаментальным объектом, представляющим текст в Java. Как мы указывали ранее, Java использует параметр `args` для передачи любых консольных аргументов, поставляемых виртуальной машине Java (VM) в вашем приложении. (Мы их не используем здесь.)

До этого момента мы свободно ссылались на переменные как на содержащие объекты. В действительности переменные, которые имеют типы классов, не столько содержат объекты, сколько указывают на них. Переменные типового класса являются ссылками на объекты. *Ссылка*

является указателем к объекту или управляет им. Если вы объявляете переменную классового типа без привязки ее к объекту, она ни на что не указывает. Она привязывается к дефолтному нолевому значению, что означает «нет значения». Если вы попробуете использовать переменную со значением `null`, как если бы она указывала на реальный объект, возникнет динамическая ошибка `NullPointerException`.

Конечно, ссылки объектов должны откуда-то браться. В нашем примере мы создали два объекта, используя операцию `new`. Мы исследуем создание объекта более подробно немного позже в этой главе.

HelloComponent

До сих пор наш пример `HelloJava` содержался в одном классе. В действительности из-за его простой природы он фактически служил как простой большой метод. Хотя мы использовали пару объектов для вывода нашего сообщения GUI, наш код не иллюстрирует никакой объектно-ориентированной структуры. Мы исправим это прямо сейчас, добавив второй класс. Чтобы от чего-то отталкиваться в этой главе, мы сменим работу класса `JLabel` (пока, `JLabel`!) и заменим его нашим собственным графическим классом `HelloComponent`. Наш класс `HelloComponent` начнется просто отображением сообщения Hello, Java! в фиксированном положении. Дальше мы добавим возможности.

Код нашего нового класса очень прост; мы добавили просто несколько новых строк:

```
import java.awt.*;
class HelloComponent extends JComponent {
public void paintComponent( Graphics g ) {
g.drawString( "Hello, Java!", 125, 95 );
}
}
```

Вы можете добавить этот текст в файл *HelloJava.java*, или вы можете поместить его в свой отдельный файл *HelloComponent.java*. Если вы поместите его в тот же файл, вы должны переместить инструкцию `new import` в самый верх файла. Чтобы использовать наш новый класс вместо `JLabel`, просто замените две строки, ссылающиеся на метку следующим кодом:

```
frame.add( new HelloComponent() );
```

Когда вы будете компилировать *HelloJava.java* в этот раз, вы увидите два бинарных классовых файла *Hello-Java.class* и *HelloComponent.class* (независимо от того, как вы организовали источник). Запуск кода будет выглядеть как версия `JLabel`, но если вы измените размер окна, вы заметите, что наш класс не регулируется автоматически для центрирования кода.

Итак, что же мы сделали и почему обратились к такой длине для надругательства над идеальным компонентом `JLabel`? Мы создали новый класс `HelloComponent`, *расширив* общий графический класс, называемый `JComponent`. Расширить класс означает добавить функционал существующему классу, создав новый. Мы изучим это в следующей главе. Здесь мы создали новый тип `JComponent`, который содержит метод, называемый `paint Component()` и ответственный за отрисовку нашего сообщения. Наш метод `paintComponent()` получает один аргумент, который называется (немного сжато) `g` и относится к типу `Graphics`. Когда вызывается метод `paintComponent()`, объект `Graphics` назначается переменной `g`, которую мы использовали в теле метода. Вскоре мы поговорим больше о `paintComponent()` и классе `Graphics`. Что же касается «почему», вы поймете, когда мы добавим новые свойства нашему новому компоненту позже.

Наследование

Классы Java организованы в иерархию, в которой родитель и ребенок известны как *надкласс* (родительский класс) и *подкласс* (дочерний класс), соответственно. Мы полностью исследуем эти понятия в главе 6. В Java у каждого класса есть только один надкласс (один родитель), но может быть много подклассов. Единственным исключением из этого правила является класс `Object`, который находится на вершине всей иерархии классов; он не имеет надклассов.

Объявление нашего класса из прошлого примера использует ключевое слово `extends`, чтобы указать, что `HelloComponent` является подклассом класса `JComponent`:

```
public class HelloComponent extends JComponent { ... }
```

Подкласс может наследовать некоторые или все переменные и методы своего надкласса. Благодаря наследованию подкласс может использовать эти переменные и методы, как будто он сам объявил их. Подкласс может добавлять свои переменные и методы, а также может *переопре-*

делить или изменить значение унаследованных методов. Когда мы используем подкласс, переопределенные методы прячутся (заменяются) своей версией подкласса для них. В этом случае наследование предоставляет мощный механизм, в результате чего подкласс может улучшать или расширять функционал своего надкласса.

Например, гипотетический класс таблиц делится на подклассы, чтобы произвести новый класс научной таблицы с дополнительными математическими функциями и специальными встроенными константами. В этом случае источник кода для научной таблицы может объявлять методы для добавленных математических функций и переменных для специальных констант, но новый класс автоматически содержит все переменные и методы, которые создают нормальную функциональность таблицы; они наследуются от родительского класса таблицы. Это также означает, что научная таблица содержит свою идентичность, как таблица, и мы можем использовать расширенную версию везде, где может использоваться более простая таблица. Последнее обстоятельство имеет далеко идущие последствия, которые мы будем изучать в книге. Это означает, что специализированные объекты могут использоваться вместо более общих объектов, настраивая свое поведение без изменения приложения, лежащего в основе. Это называется *полиморфизмом* и является одной из основ объектно-ориентированного программирования.

Наш класс `HelloComponent` является подклассом класса `JComponent` и наследует многие переменные и методы, явно объявленные в нашем исходном коде. Это то, что позволяет нашему маленькому классу служить компонентом `JFrame` всего лишь с несколькими настройками.

Класс `JComponent`

Класс `JComponent` предоставляет фреймворк для создания всех видов компонентов интерфейса пользователя. Отдельные компоненты — такие, как кнопки, метки и списковые окна — реализованы как подклассы `JComponent`.

Мы переопределяем методы в таких подклассах, чтобы обеспечить поведение нашего отдельного компонента. Это может звучать, как будто мы ограничены каким-то предопределенным набором рутинных действий, но все совсем не так. Запомните, что методы, о которых мы говорим, являются способами взаимодействия с оконной системой. Нам не нужно ограничивать все наше приложение. Реальное приложение долж-

но включать сотни или тысячи классов с множеством методов и переменных и многими потоками выполнения. Подавляющее большинство их связано с частностями нашей работы (они называются объектами *домена*). Класс `JComponent` и другие предопределенные классы служат только фреймворком, где размещается код, который содержит определенные типы событий интерфейса пользователя и показывает информацию пользователю.

Метод `paintComponent()` является важным методом класса `JComponent`; мы переопределяем его, чтобы применить способ, которым наш отдельный компонент выводится на экране. Поведение по умолчанию `paintComponent()` не производит никакого рисования. Если бы мы не переопределили его в нашем подклассе, наши компоненты были бы просто невидимыми. Мы переопределяем `paintComponent()`, чтобы создать что-то хоть немного более интересное. Мы не переопределяем никакие другие унаследованные члены `JComponent`, потому что они предоставляют базовую функциональность и приемлемые свойства по умолчанию для этого (пустякового) примера. По мере расширения `HelloJava` мы углубимся в унаследованные элементы и будем использовать дополнительные методы. Мы также добавим некоторые специфичные для приложения методы и переменные специально для нужд `HelloComponent`.

`JComponent` фактически является верхушкой айсберга, называемого `Swing`. `Swing` — это набор инструментов для интерфейса пользователя, представленный в нашем примере инструкцией `import` в самом верху; мы обсудим его в некоторых деталях в главах с 16–18.

Связи и поиск виновных

Мы можем верно сослаться на `HelloComponent` как на `JComponent`, поскольку создание подклассов можно воспринимать как создание отношения «является», в котором подкласс «является» чем-то типа его надкласса. Таким образом, `HelloComponent` является видом `JComponent`. Когда мы обращаемся к типу объекта, мы имеем в виду любое представление класса этого объекта или каких-либо его подклассов. Далее мы рассмотрим поближе иерархию классов Java и увидим, что `JComponent` сам является подклассом класса `Container`, который в свою очередь происходит от класса, называемого `Component`, и т. д., как показано на рис. 2.6.

В этом смысле объект `HelloComponent` является видом `JComponent`, который является видом `Container`, и каждый из них может в итоге счи-

таться видом Component. Именно от этих классов HelloComponent наследует свою базовую функциональность GUI и (как мы обсудим позже) способность вставлять графические компоненты.

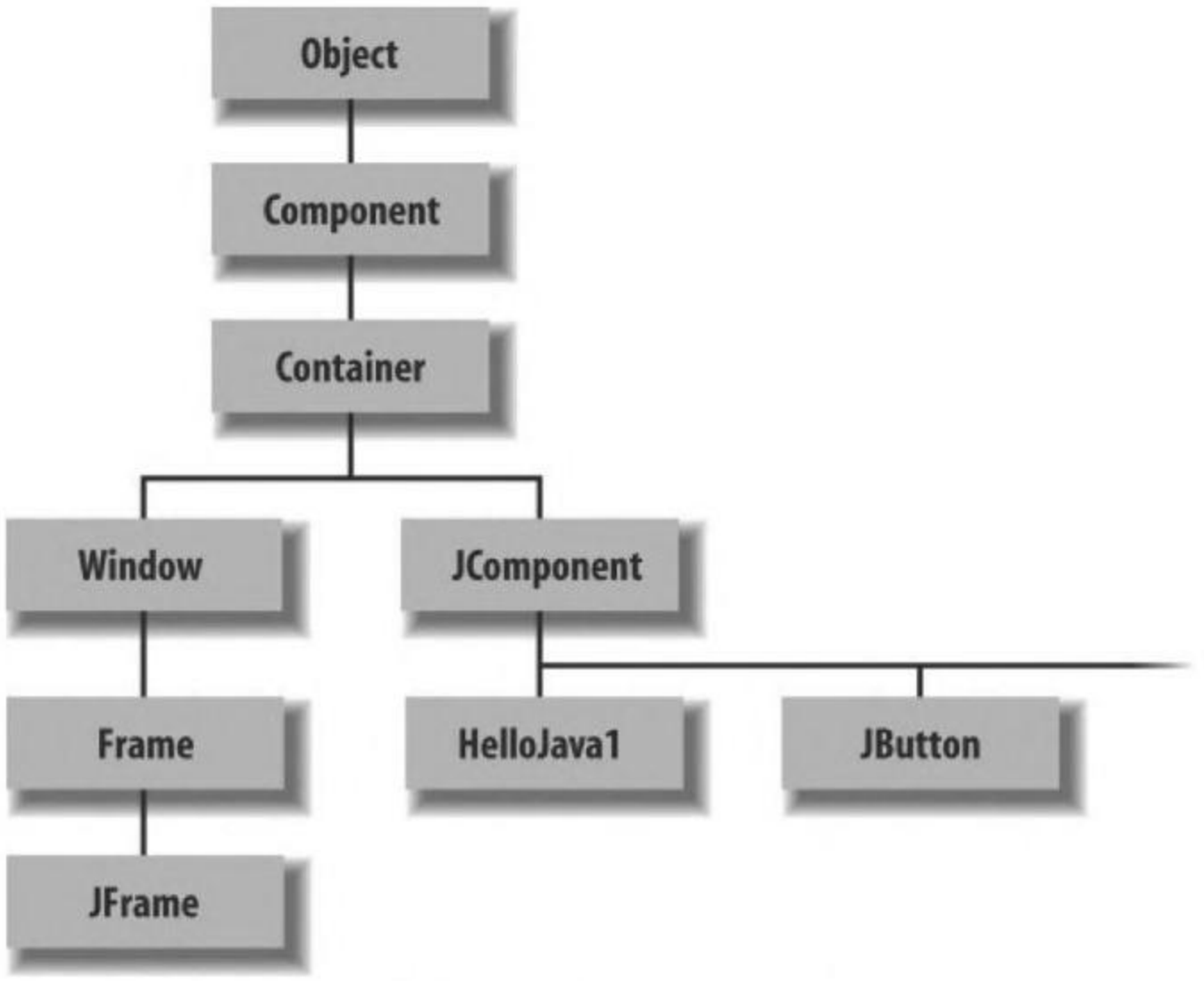

Рис. 2.6. Часть иерархии классов Java

Component является подклассом класса Object высшего уровня, поэтому все эти классы являются типами Object. Каждый последующий класс интерфейса Java API наследует поведение от класса Object, который определяет некоторые базовые методы, что вы увидите в главе 7. Мы продолжим использовать слово *объект* со строчной «о» в общем смысле, чтобы сослаться на проявление любого класса; мы будем использовать Object, чтобы сослаться сугубо на тип этого класса.

Пакеты и инструкции импорта

Ранее мы упоминали, что первая строка нашего примера говорит Java, где найти некоторые из классов, которые мы использовали:

```
import javax.swing.*;
```

Если быть точнее, он говорит компилятору, что мы собираемся использовать классы набора инструментов Swing GUI (в данном случае

JFrame, JLabel и JComponent). Эти классы организованы в *пакете* Java, называемом javax.swing. Пакет Java является группой классов, которые связаны целью или приложением. Классы в одном пакете имеют специальные привилегии доступа относительно друг друга и могут быть спроектированы для совместной тесной работы.

Пакеты именуются в иерархическом стиле с компонентами, разделенными точкой, например java.util и java.util.zip. Классы в пакете должны следовать конвенции о том, где они должны быть расположены в путях к классу. Они также принимают имя пакета в качестве части их «полного имени» или, если использовать правильную терминологию, их *полное квалифицированное имя*. Например, полное квалифицированное имя класса JComponent — javax.swing.JComponent. мы можем ссылаться к нему прямо по этому имени вместо использования инструкции import:

```
public class HelloComponent extends javax.swing.JComponent {...}
```

Инструкция import javax.swing.* позволяет нам обратиться ко всем классам пакета javax.swing по их простым именам. Поэтому нам необязательно использовать полное квалифицированное имя, чтобы обратиться к классам JComponent, JLabel и JFrame.

Как мы видели, когда добавляли класс для нашего второго примера, в заданном исходном файле Java может быть один или более операторов import. Инструкции import эффективно создают «путь поиска» файлов, говорящий Java, где искать классы, к которым мы обращаемся по их простым, неквалифицированным именам. (Это в действительности не является путем, но позволяет избежать сомнительных имен, которые могут создать ошибки.) Инструкции import, которые мы видели, используют обозначение с точкой и звездочкой (.*), чтобы указать, что должен быть импортирован весь пакет. Но вы также можете указать и один класс. Наш текущий пример использует только класс Graphics пакета java.awt package. Так что мы могли бы использовать инструкцию import java.awt.Graphics вместо использования символа обобщения * для импорта всех классов пакета абстрактно-оконного инструментария (AWT). Однако мы ожидаем использование еще нескольких классов этого пакета позже.

Иерархии пакетов java. и javax. особенные. Любой пакет, который начинается с java. является частью базового интерфейса Java API и доступен на любой платформе, которая поддерживает Java. Пакет javax. обычно обозначает стандартное расширение основной платформы, которое может быть установлено или не установлено. Однако за последние годы многие стандартные расширения были добавлены в базовый

интерфейс Java API без переименования. Пакет javax.swing является примером; это часть базового интерфейса API, несмотря на его имя. Рис. 2.7 иллюстрирует некоторые базовые пакеты Java, показывая типичный класс или два из каждого.

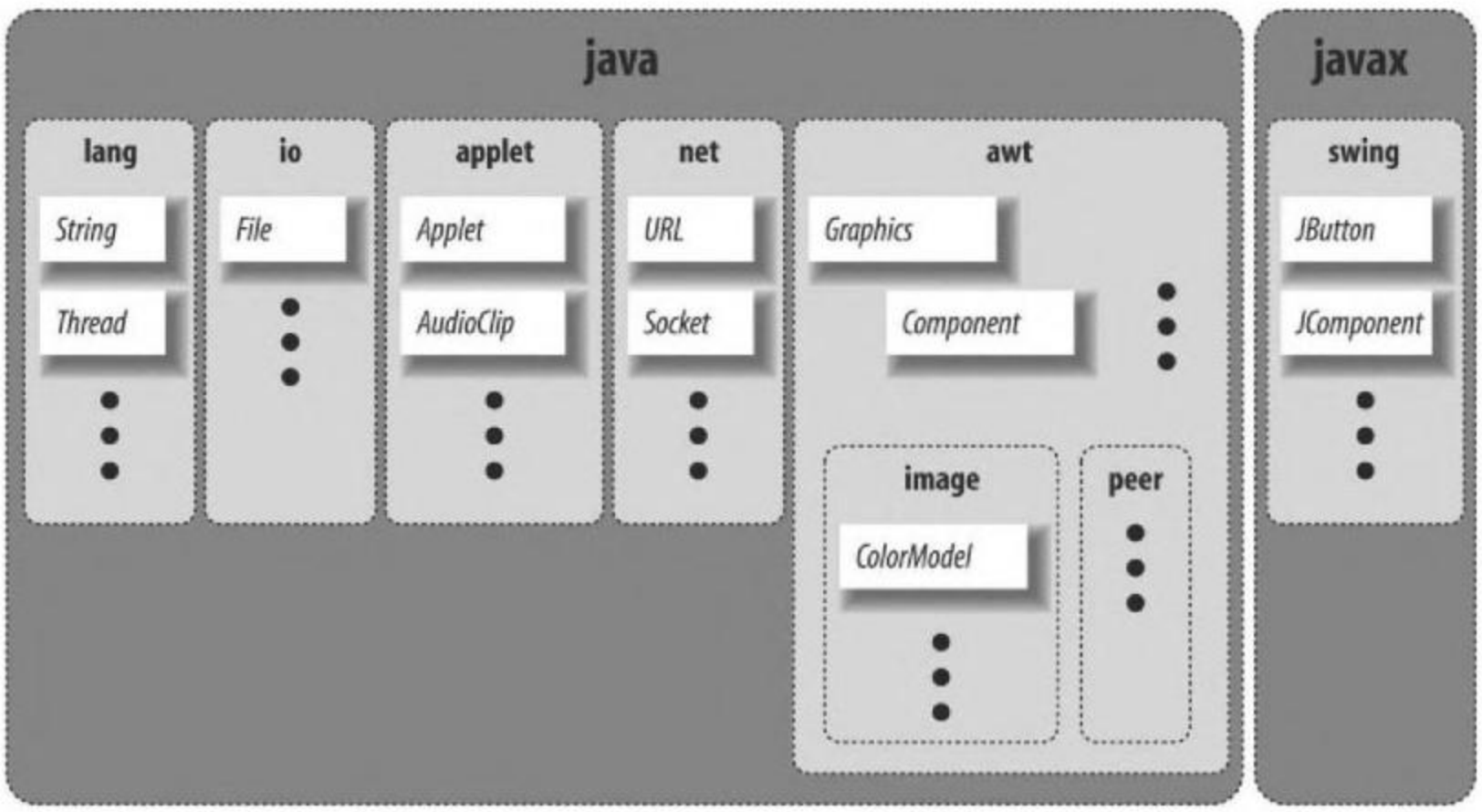

Рис. 2.7. Некоторые базовые пакеты Java

Пакет java.lang содержит базовые классы, необходимые самому языку Java; этот пакет импортируется автоматически, и поэтому нам не нужна инструкция импорта для использования таких имен классов, как String или System в наших примерах. Пакет java.awt содержит классы старшего абстрактно-оконного инструментария; java.net содержит сетевые классы и т. д. По мере приобретения опыта вы поймете, что и наличие команд пакетов, доступных вам, и то, что они делают, и когда их использовать, и как их использовать — все это важно для превращения в успешного разработчика на языке Java.

Метод paintComponent()

Источник нашего класса HelloComponent определяет метод, paintComponent(), который перезаписывает метод paintComponent() класса JComponent:

```
public void paintComponent( Graphics g ) {
g.drawString( "Hello, Java!", 125, 95 );
}
```

Метод `paintComponent()` вызывается, когда нашему примеру приходит время нарисовать себя на экране. Он использует один аргумент, объект класса `Graphics`, и не возвращает никаких типов и значения (`void`) тому, кто его вызывает.

Модификаторы — это ключевые слова, которые ставятся перед классами, переменными и методами, чтобы менять их доступность, поведение или семантику. `paintComponent()` объявляется публичным, и это означает, что он может вызываться методами в классах, отличных от `HelloComponent`. В этом случае оконная среда Java вызывает наш метод `paintComponent()`. Метод или переменная, объявленная как `private`, доступна только из своего класса.

Объект класса `Graphics`, представитель класса `Graphics`, представляет частную графическую область для рисования. (Она также называется *графическим контекстом*.) Он содержит методы, которые могут использоваться для рисования в этой области, и переменные, которые представляют такие характеристики, как режим вырезания или рисования. Отдельный объект класса `Graphics`, который мы передаем методу `paintComponent()`, соответствует области `HelloComponent` нашего экрана внутри нашего фрейма.

Класс `Graphics` предоставляет методы для визуализации форм, изображений и текста. В `HelloComponent` мы вызываем метод `drawString()` нашего объекта класса `Graphics`, чтобы небрежно написать наше сообщение в специфических координатах. (Для описания методов, доступных в классе `Graphics`, смотрите главу 20.)

Как мы видели ранее, мы получаем доступ к методу объекта прибавляя точку (.) и его имя к объекту, который содержит его. Мы вызвали метод `drawString()` объекта `Graphics` (к которому ссылается наша переменная `g`) таким образом:

```
g.drawString( "Hello, Java!", 125, 95 );
```

Может быть, сложно привыкнуть к идее того, что наше приложение нарисовано методом, который вызывается внешним агентом в любой момент времени. Как мы можем с пользой это применить? Как мы контролируем, что и когда выполняется? Ответы на эти вопросы нам предстоит узнать. На данный момент просто подумайте о том, как вы начнете строить приложения, которые отвечают на команды, а не действуют по собственной инициативе.

HelloJava2: продолжение

Теперь, когда у нас есть некая база, давайте сделаем наше приложение немного более интерактивным. Следующее небольшое улучшение позволяет нам двигать текстовое сообщение при помощи мыши.

Мы назовем этот пример HelloJava2, чтобы не вводить путаницу, продолжая расширять старый пример. Но основные дополнения здесь и далее заключаются в добавлении возможностей к классу HelloComponent и просто в соответствующем изменении имен, чтобы не запутаться (например, HelloComponent2, HelloComponent3 и т. д.). Увидев наследование в действии, вы, возможно, удивитесь, почему мы не создаем подклассы для HelloComponent и не эксплуатируем наследование, чтобы основать на нем наш предыдущий пример и расширить его функциональность. Что ж, в нашем случае это не даст большого преимущества, и для ясности мы просто начнем с начала.

Ниже представлен код приложения HelloJava2:

```
//файл: HelloJava2.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HelloJava2
{
public static void main( String[] args ) {
JFrame frame = new JFrame( "HelloJava2" );
frame.add( new HelloComponent2("Hello, Java!") );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setSize( 300, 300 );
frame.setVisible( true );
}
}
class HelloComponent2 extends JComponent
implements MouseMotionListener
{
String theMessage;
int messageX = 125, messageY = 95; // Координаты сообщения
public HelloComponent2( String message ) {
theMessage = message;
addMouseMotionListener(this);
}
```

```
public void paintComponent( Graphics g ) {
g.drawString( theMessage, messageX, messageY );
}
public void mouseDragged(MouseEvent e) {
// Сохранить координаты мыши и нарисовать сообщение.
messageX = e.getX();
messageY = e.getY();
repaint();
}
public void mouseMoved(MouseEvent e) { }
}
```

Две наклонные черты в строке означают, что остальная часть строки является комментарием. Мы добавили несколько комментариев к `HelloJava2`, чтобы помочь вам следить за происходящим.

Поместите текст этого примера в файл с названием *HelloJava2.java* и скомпилируйте, как прежде. В результате вы должны получить новые файлы классов, *HelloJava2.class* и *HelloComponent2.class*.

Запустите пример, используя следующую команду:

```
C:\> java HelloJava2
```

Или если вы используете Eclipse, нажмите кнопку **Run** (Выполнить). Свободно замещайте своими непристойными комментариями сообщение Hello, Java! и наслаждайтесь часами веселья, перетаскивая мышью текст. Заметьте, что сейчас, когда вы нажимаете кнопку закрытия окна, приложение закрывается; мы объясним это позже, когда будем говорить о событиях. Теперь давайте посмотрим, что изменилось.

Переменные экземпляра

Мы добавили некоторые переменные в класс `HelloComponent2` в нашем примере:

```
int messageX = 125, messageY = 95;
String theMessage;
```

`messageX` и `messageY` являются целыми значениями, которые хранят координаты нашего двигающегося сообщения. Мы кое-как связали их с дефолтными значениями, которые должны поместить сообщение где-то недалеко от центра окна. Целые значения Java — 32-битные относи-

тельные числа, поэтому они легко могут хранить значения наших координат. Переменная `theMessage` относится к типу `String` и может хранить экземпляр класса `String`.

Вы должны принять во внимание, что эти три переменные объявляются внутри скобок определения класса, но не внутри какого-либо метода класса. Эти переменные называются *переменными экземпляра* и принадлежат всему классу. Так, их копии появляются в каждом отдельном экземпляре класса. Переменные экземпляра всегда видимы (и используются) всеми методами внутри своего класса. В зависимости от их модификаторов они также могут быть доступны вне класса.

Переменные экземпляра имеют дефолтное значение `0`, `false` или `null`, в зависимости от их типа, если не присвоено другое значение. Числовые типы имеют значение `0`, булевы переменные — `false`, а переменные типа класса всегда имеют значение `null`, то есть «нет значения». Попытка использовать объект с нолевым значением приведет к ошибке выполнения.

Переменные экземпляра отличаются от аргументов метода и других переменных, объявленных внутри рамок отдельного метода. Последние называются *локальными* переменными. Они являются, по сути, приватными переменными, которые видны только коду внутри метода. Java не устанавливает локальные переменные, поэтому вы можете назначить их сами. Если вы используете переменную, которой еще не было назначено значение, ваш код сгенерирует ошибку во время компиляции. Локальные переменные существуют, только пока выполняется метод, а затем исчезают, если что-то другое не спасет их значение. Каждый раз, когда вызывается метод, его локальные переменные восстанавливаются и должны иметь приписанное значение.

Мы использовали новые переменные, чтобы сделать наш ранее громоздкий метод `paintComponent()` более динамическим. Теперь все аргументы в вызове `drawString()` определены этими переменными.

Конструкторы

Класс `HelloComponent2` включает специальный тип метода, называемый *конструктор*. Конструктор вызывается для создания нового экземпляра класса. Когда создается новый объект, Java назначает ему хранилище, устанавливает переменным значениям их дефолтные зна-

чения и вызывает метод-конструктор для класса, чтобы выполнить то, чего требует установка на уровне приложения.

Конструктор всегда имеет то же имя, что и класс. Например, конструктор класса `HelloComponent2` называется `HelloComponent2()`. Конструкторы не имеют возвращаемого типа, но вы можете считать их создающими объект типа своего класса. Как и другие методы, конструкторы могут иметь аргументы. Их единственная миссия — конфигурировать и инициализировать только что созданные экземпляры класса, возможно, используя информацию, передаваемую им в этих параметрах.

Объект создается операцией `new`, указывающей конструктор для класса и необходимые аргументы. Получившийся в результате экземпляр объекта возвращается как значение. В нашем примере новый экземпляр `HelloComponent2` создается методом `main()` этой строкой:

```
frame.add( new HelloComponent2("Hello, Java!") );
```

Эта строка в действительности делает две вещи. Мы можем написать их как две отдельные строки, что будет немного проще понять:

```
HelloComponent2 newObject = new HelloComponent2("Hello, Java!");
frame.add( newObject );
```

Первая строка является важной, в ней создается новый объект `HelloComponent2`. Конструктор `HelloComponent2` берет класс `String` в качестве аргумента и, когда мы его закрепляем, использует его для создания сообщения, которое показывается в окне. При помощи небольшой магии компилятора Java текст в кавычках в исходном коде Java превращается в объект класса `String`. (Смотрите главу 10, где подробно обсуждается класс `String`.) Вторая строка просто добавляет наш новый компонент во фрейм, чтобы сделать его видимым, как происходило и в предыдущих примерах. Если вы хотите сделать свое сообщение гибким, можете изменить строку конструктора на следующую:

```
HelloComponent2 newobj = new HelloComponent2( args[0] );
```

Теперь вы можете передать текст в командную строку, когда запустите приложение, используя следующую команду:

```
C:\> java HelloJava2 "Hello, Java!"
```

`args[0]` относится к первому параметру командной строки. Его значение станет более понятным, когда мы будем обсуждать массивы далее

в этой книге. Если вы используете среду IDE вроде Eclipse, вы должны будете настроить ее так, чтобы принимать ваши параметры перед ее запуском.

Конструктор `HelloComponent2` затем делает две вещи: устанавливает текст переменной экземпляра `theMessage` и вызывает метод `addMouseMotionListener()`. Этот метод является частью механизма события, который мы обсудим далее. Он говорит системе: «Эй, мне интересно все, что происходит с мышью».

```
public HelloComponent2(String message) {
theMessage = message;
addMouseMotionListener( this );
}
```

Специальная переменная только для чтения, именуемая `this`, используется для однозначного обращения к нашему объекту («текущий» контекст объекта) в вызове `addMouseMotionListener()`. Метод может использовать `this`, чтобы сослаться на экземпляр объекта, который ее содержит. Таким образом, следующие две инструкции являются эквивалентными способами назначить значение переменной экземпляра `theMessage`:

```
theMessage = message;
```

или:

```
this.theMessage = message;
```

Обычно мы используем более короткую, имплицитную форму для ссылки на переменные экземпляра, но нам понадобится `this`, когда необходимо будет эксплицитно передать ссылку на наш объект методу другого класса. Мы часто делаем это так, что методы других классов могут вызывать наши публичные методы или использовать наши публичные переменные.

События

Последние два метода `HelloComponent2`, `mouseDragged()` и `mouseMoved()`, позволяют нам получать информацию от мыши. Каждый раз, когда пользователь совершает действие, например нажимает клавишу на клавиатуре, двигает мышь или бьется головой о сенсорную панель, Java генерирует *событие*. Событие представляет собой действие, кото-

рое произошло; оно содержит информацию о действии, например время и место. Многие события ассоциируются с отдельным компонентом GUI приложения. Нажатие на клавишу, например, может соответствовать символу, введенному в определенное поле ввода текста. Нажатие на кнопку мыши может активировать определенную кнопку на экране. Даже движение мыши по определенной области экрана может запустить такие эффекты, как подсветка или изменение формы курсора.

Чтобы работать с этими событиями, мы импортировали новый пакет `java.awt.event`, который предоставляет специфические объекты событий для получения информации от пользователя. (Заметьте, что импорт `java.awt.*` автоматически не импортирует пакет событий. Пакеты обычно не содержат другие пакеты, даже если иерархическая схема имен предполагает, что это так.)

Существует много разных событий класса, включая `MouseEvent`, `KeyEvent` и `ActionEvent`. По большей части значения этих событий интуитивно понятны. `MouseEvent` возникает, когда пользователь что-то делает при помощи мыши, `KeyEvent` возникает, когда пользователь нажимает на клавишу. и т. д. `ActionEvent` — немного особенный. Мы увидим его в действии далее в этой главе, в третьей версии нашего `HelloJava`. Сейчас мы сосредоточимся на работе с `MouseEvents`.

Компоненты GUI в Java создают события для специфических видов деятельности пользователя. Например, если вы щелкаете мышью внутри компонента, компонент сгенерирует событие мыши. Объекты могут запрашивать получение событий от одного или более компонентов, регистрируя *приемник* с источником событий. Например, чтобы объявить, что приемник хочет получить событие компонента на движение мыши, вы вызываете метод `addMouseMotionListener()` этого компонента, указывая, что объект приемника является аргументом. Это то, что наш пример делает в своем конструкторе. В этом случае компонент вызывает свой собственный метод `addMouseMotionListener()` с аргументом `this`, что подразумевает «Я хочу получить свои собственные события на движение мыши». Так мы настраиваем получение событий. Но как мы в действительности получаем их? Это как раз то, для чего нужны нам два относящихся к мыши метода в классе. Метод `mouseDragged()` вызывается автоматически на приемнике для получения событий, генерирующихся, когда пользователь перемещает мышь, то есть двигает мышь с любой зажатой кнопкой. Метод `mouseMoved()` вызывается всегда, когда пользователь двигает мышью по области, не нажимая на кнопку. В этом случае мы поместили эти методы в наш класс `HelloComponent2` и позволили ему

настроить себя как приемник. Это полностью подходит нашему новому компоненту с перетаскиванием текста. Более обобщенно, хороший дизайн обычно диктует, что приемники событий должны внедряться как *классы адаптера*, которые предоставляют лучшее разделение GUI и «бизнес-логики». Мы обсудим это более подробно далее в этой книге.

Наш метод `mouseMoved()` скучный: он ничего не делает. Мы игнорируем простые движения мыши и приберегаем наше внимание для перемещения. У метода `mouseDragged()` больше пищи для работы над этим. Этот метод вызывается повторно оконной системой, чтобы мы получали свежие данные по положению мыши. Ниже представлен код.

```
public void mouseDragged( MouseEvent e ) {
messageX = e.getX();
messageY = e.getY();
repaint();
}
```

Первый аргумент `mouseDragged()` — объект `MouseEvent`, `e`, который содержит всю необходимую нам информацию об этом событии. Мы просим `MouseEvent` сообщать нам координаты x и y текущего положения мыши, вызывая методы `getX()` и `getY()`. Мы сохраняем эти данные в переменных экземпляра `messageX` и `messageY` для использования где бы то ни было.

Красота модели событий заключается в том, что вы можете управлять только нужными типами событий. Если вам неинтересны события клавиатуры, вы просто не регистрируете приемник для них; пользователь может набирать все, что ему вздумается, но вас это не будет волновать. Если нет приемника для отдельного типа событий, Java не будет даже их генерировать. В результате управление событиями является довольно эффективным*.

Пока мы обсуждаем события, мы должны упомянуть еще одно маленькое дополнение, которое упустили в `HelloJava2`:

```
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

Эта строка говорит фрейму закрыть приложение, когда нажата кнопка выхода. Это называется операцией закрытия по умолчанию, поскольк-

---

* Обработка событий в Java 1.0 была совсем другой историей. Ранее язык Java не имел понятия о приемниках событий и вся обработка событий осуществлялась путем перезаписи методов в базовых классах GUI. Это было в равной степени неэффективно и вело к сырому проекту с распространением высоко специализированных компонентов.

ку эта операция, как почти любое другое взаимодействие с GUI, управляется событием. Мы могли бы зарегистрировать окно приемника, чтоб получать уведомление о том, когда пользователь нажимает на кнопку закрытия, и производить любое действие, которое нам нравится, но этот удобный метод управляет общими процессами.

В итоге мы увильнули от пары вопросов: как наша система узнает, что наш класс содержит необходимые методы `mouseDragged()` и `mouseMoved()` (откуда происходят эти имена)? И почему мы должны поддерживать метод `mouseMoved()`, который ничего не делает? Ответ на эти вопросы имеет отношение к интерфейсам. Мы обсудим интерфейсы после того, как завершим некоторые незаконченные дела с `repaint()`.

Метод `repaint()`

Поскольку мы изменили координаты сообщения (когда перетаскивали мышь), мы бы хотели, чтобы `HelloComponent2` перерисовался. Мы осуществляем это при помощи вызова `repaint()`, который просит систему перерисовать экран через какое-то время. Мы не можем прямо обратиться к `paintComponent()`, даже если бы мы этого хотели, поскольку у нас нет графического содержимого, чтоб передать ему.

Мы можем использовать метод `repaint()` класса `JComponent`, чтобы потребовать перерисовки нашего компонента. `repaint()` вынуждает оконную систему Java включить в план запрос к нашему методу `paintComponent()` в ближайшее возможное время; Java предоставляет необходимый объект класса `Graphics`, как показано на рис. 2.8.

Этот режим выполнения — не просто неудобство, вызванное неимением подходящего графического контекста под рукой. Главное преимущество этого режима — то, что поведение перерисовки управляется кем-то другим, в то время как мы можем заниматься своими делами. Система Java имеет отдельный, выделенный поток выполнения, который управляет всеми запросами `repaint()`. Он может упорядочивать и объединять запросы `repaint()`, как это необходимо, что позволяет предотвратить перезагрузку оконной системы во время ситуаций с интенсивным рисованием, таких как прокрутка. Другое преимущество — это то, что весь функционал рисования должен быть воплощен в наш метод `paintComponent()`; нас не прельщает распространять его по всему приложению.

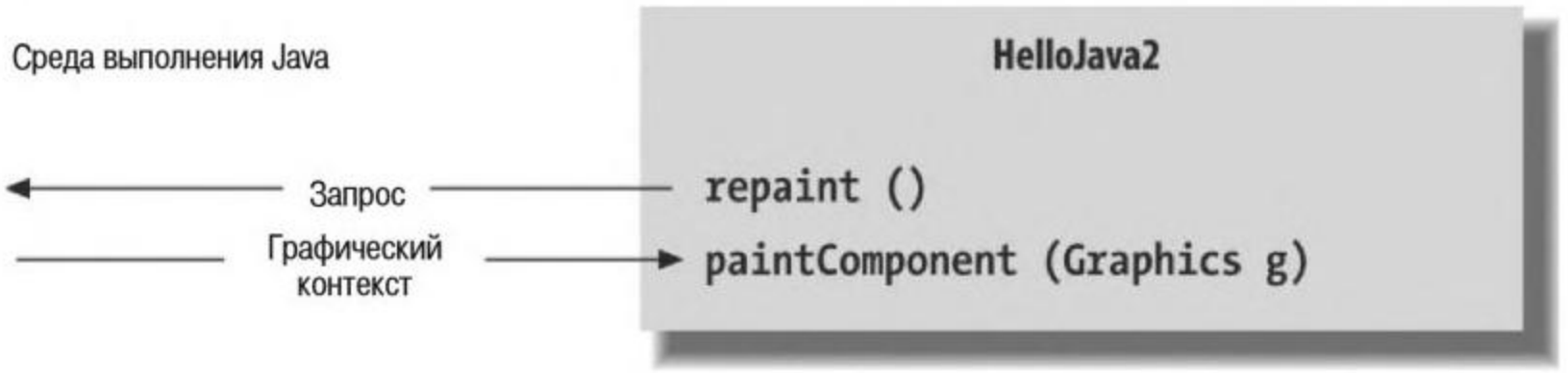

Рис. 2.8. Вызов метода `repaint()`

Интерфейсы

Теперь пора столкнуться с вопросами, которых мы избегали ранее: как система понимает, что нужно вызвать `mouseDragged()`, когда возникает событие мыши? Может, `mouseDragged()` — это просто какое-то магическое имя, которое должен иметь наш метод управления событиями? Не совсем; ответ на этот вопрос затрагивает тему интерфейсов, которые являются одним из важнейших свойств языка Java. Первый признак интерфейса проступает на строчке кода, которая вводит класс `HelloComponent2`: мы говорим, что класс реализует интерфейс `MouseMotionListener`.

```
class HelloComponent2 extends JComponent
implements MouseMotionListener
{
```

Изначально, интерфейс — список методов, которые должен иметь класс; этот отдельный интерфейс требует, чтобы у нашего класса были методы, называемые `mouseDragged()` и `mouseMoved()`. Интерфейс не говорит, что должны делать эти методы; в действительности `mouseMoved()` ничего не делает. Он говорит, что методы должны использовать `MouseEvent` как аргумент и возвращать пустое значение (вот что означает `void`).

Интерфейс — контракт между вами, разработчиком кода и компилятором. Говоря, что ваш класс реализует интерфейс `MouseMotionListener`, вы заявляете, что эти методы будут доступны для вызова другими частями системы. Если вы не предоставите их, возникнет ошибка компиляции.

Это не единственный способ влияния интерфейсов на программу. Интерфейс также ведет себя как класс. Например, метод может вернуть `MouseMotionListener` или использовать `MouseMotionListener` как аргумент. Когда вы ссылаетесь к объекту по имени интерфейса таким способом, это означает, что вам не важен действительный класс объекта; единственное требование — это то, что класс реализует этот интерфейс. `addMouseMotionListener()` является таким методом: его аргумент должен быть объектом, который реализует интерфейс `MouseMotionListener`.

Аргумент, который мы передаем `this`, — сам объект `HelloComponent2`. Факт, что он экземпляр `JComponent` не имеет значения; это может быть `Cookie`, `Aardvark` или любой другой класс, который мы придумаем. Важно лишь то, что он применяет `MouseMotionListener` и, таким образом, объявляет, что будет иметь два именованных объекта. Поэтому нам нужен метод `mouseMoved()`, пусть даже тот, который мы предоставили, ничего не делает: `MouseMotionLisTener` говорит, что он нам нужен.

Дистрибутив Java поступает со многими интерфейсами, которые определяют, что должны делать классы. Эта идея контракта между компилятором и классом очень важна. Существует много ситуаций, как та, которую мы только что видели, где вам неважно, чем является класс, вам важно, что у него есть такая способность, как прием событий мыши. Интерфейсы дают нам возможность работать над объектами, основываясь на их возможностях, не зная или не обращая внимания на их действительный тип. Интерфейсы являются важным понятием для использования Java в качестве объектно-ориентированного языка, и мы подробнее поговорим о них в главе 4. Как мы вскоре увидим, интерфейсы добавляют оговорку к правилу Java, что любой новый класс может расширить только один класс («единственное наследование»). Класс в языке Java может расширить только один класс, но может реализовать сколько угодно интерфейсов; наш следующий пример реализует два интерфейса, а последний пример в этой главе — три. Во многом интерфейсы очень похожи на классы, но не совсем. Они могут быть использованы как типы данных, могут расширять другие интерфейсы (но не классы) и могут наследоваться классами (если класс A внедряет интерфейс B, подклассы A также внедряют B). Принципиальная разница состоит в том, что классы в действительности не наследуют методы у интерфейсов; интерфейсы просто уточняют методы, которые должен иметь класс.

`HelloJava3`: атака класса `JButton`!

Теперь мы можем перейти к более веселым вещам. `HelloJava3` принес нам новый компонент графического интерфейса: `JButton`*. В этом примере мы добавляем к нашему приложению компонент `JButton`, кото-

* Почему не просто `Button`? `Button` — это имя, которое использовалось в оригинальном наборе инструментов GUI Java, AWT. В инструментарии AWT были некоторые незначительные недостатки, поэтому он был расширен и фактически заменен Swing в Java 1.2. Поскольку AWT уже занял приемлемые имена, такие как `Button` и `MenuBar`, и смешивание их в коде могло сбить с толку, имена компонентов интерфейса пользователя в Swing начинаются с J, как, например, `JButton` и `JMenuBar`.

рый меняет цвет нашего текста каждый раз, когда нажимается кнопка. Возможность перемещать сообщения также все еще там. Наш новый код выглядит так:

```
//файл: HelloJava3.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HelloJava3
{
public static void main( String[] args ) {
JFrame frame = new JFrame( "HelloJava3" );
frame.add( new HelloComponent3("Hello, Java!") );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setSize( 300, 300 );
frame.setVisible( true );
}
}
class HelloComponent3 extends JComponent
implements MouseMotionListener, ActionListener
{
String theMessage;
int messageX = 125, messageY = 95; // Координаты сообщения
JButton theButton;
int colorIndex; // текущий указатель в объекте someColors
static Color[] someColors = {
Color.black, Color.red, Color.green, Color.blue, Color.magenta };
public HelloComponent3( String message ) {
theMessage = message;
theButton = new JButton("Change Color");
setLayout( new FlowLayout() );
add( theButton );
theButton.addActionListener( this );
addMouseMotionListener( this );
}
public void paintComponent( Graphics g ) {
g.drawString( theMessage, messageX, messageY );
}
public void mouseDragged( MouseEvent e ) {
messageX = e.getX();
messageY = e.getY();
```

```
repaint();
}
public void mouseMoved( MouseEvent e ) {}
public void actionPerformed( ActionEvent e ) {
// Кто-то нажимал нашу кнопку?
if (e.getSource() == theButton)
changeColor();
}
synchronized private void changeColor() {
// Изменить индекс на другой цвет, неудобно.
if (++colorIndex == someColors.length)
colorIndex = 0;
setForeground( currentColor() ); // Использовать новый цвет.
repaint();
}
```

Скомпилируйте `HelloJava3` так же, как и другие приложения. Запустите приложение, и вы увидите экран, показанный на рис. 2.9. Переместите текст. Каждый раз, когда вы нажимаете на кнопку, цвет должен изменяться.



Рис. 2.9. Приложение `HelloJava3`

Что мы добавили сейчас? Что ж, для начала у нас новая переменная:

```
JButton theButton;
```

Переменная `theButton` относится к типу `JButton` и хранит экземпляр класса `javax.swing.JButton`. Класс `JButton`, как вы можете ожидать, представляет графическую кнопку, такую же, как другие кнопки в вашей оконной системе.

Три дополнительные строки в конструкторе создают кнопку и показывают ее:

```
theButton = new JButton("Change Color");
setLayout( new FlowLayout() );
add( theButton );
```

В первой строке ключевое слово `new` создает экземпляр класса `JButton`. Следующая строчка влияет на то, как наш компонент будет использовать контейнер, чтобы хранить кнопку. Он сообщает `HelloComponent3`, как должны быть организованы компоненты, которые добавляются к нему для демонстрации — в этом случае для использования схемы, называемой `FlowLayout` (об этом поговорим позже). В итоге он добавляет кнопку к нашему компоненту, как мы добавляли `HelloComponent3` к окну содержимого `JFrame` в методе `main()`.

Перегрузка метода

У `JButton` больше одного конструктора. У класса множество конструкторов, каждый принимает разные параметры и, вероятно, использует их для произведения разных видов установки. Когда у класса много конструкторов, Java выбирает один правильный, основанный на типах аргументов, используемых с ним. Мы вызываем конструктор `JButton` с аргументом `String`, поэтому Java определяет метод конструктора класса `JButton`, который берет один аргумент `String` и использует, чтобы установить объект. Это называется *перегрузка метода*. Все методы в Java (не только конструкторы) могут перегружаться; это еще один аспект принципа *полиморфизма* в объектно-ориентированном программировании.

Перегруженные конструкторы в основном предоставляют удобный путь инициализации нового объекта. Конструктор `JButton`, который мы использовали для установки текста кнопки, когда он создается:

```
theButton = new JButton("Change Color");
```

Вот сокращенная запись создания кнопки и установки ярлыка:

```
theButton = new JButton();
theButton.setText("Change Color");
```

Компоненты

Мы использовали понятия *компонент* и *контейнер* немного неточно, чтобы описать графические элементы Java-приложения, но эти понятия применяются в именах настоящих классов в пакете `java.awt`.

`Component` — это базовый класс, из которого происходят все компоненты интерфейса GUI Java. Он содержит переменные, которые пред-

ставляют положение, форму, общий вид и статус объекта и методов для общего рисования и обработки событий. avax.swing.JComponent расширяет базовый класс Component и улучшает его для набора инструментов Swing. Метод paintComponent(), который мы использовали в нашем примере, унаследован от класса JComponent. HelloComponent — это вид JComponent и наследует все его публичные члены, как другие компоненты GUI.

Класс JButton также производится от JComponent и, таким образом, разделяет эту функциональность. Это значит, что разработчик класса JButton имел в наличии такие методы, как paintComponent(), с помощью которых он мог реализовать поведение объекта JButton, как мы делали, создавая наш пример. Что удивительно, так это то, что мы абсолютно свободны создавать дальнейшие компоненты подкласса, такие как JButton, и перезаписывать их поведение, чтобы создать собственные типы компонентов интерфейса пользователя. JButton и HelloComponent3 в этом отношении являются эквивалентными типами.

Контейнеры

Класс Container является расширенным типом класса Component, который содержит список дочерних компонентов в иерархическом виде и помогает группировать их. Container заставляет своих детей показываться и упорядочивает их на экране согласно отдельной стратегии разработки. Поскольку класс Container также является классом Component, он может быть помещен рядом с другими объектами Component в других классах Container иерархически, как показано на рис. 2.10. Наш класс HelloComponent3 является типом Container (в силу класса JComponent) и, таким образом, может хранить другие компоненты и контейнеры Java, такие как кнопки, ползунковые регуляторы, текстовые поля и панели, а также управлять ими. На рис. 2.10 элементы, обозначенные курсивом, являются объектами класса Component, а обозначенные жирным текстом — Container. Кнопочная панель реализована как объект контейнера, который управляет рядом кнопок. Сама кнопочная панель содержится в объекте контейнера GizmoTool.

Поскольку JComponent происходит от Container, он может быть и компонентом, и контейнером. Фактически мы уже использовали его в такой роли в примере HelloComponent3. Он производит свое собственное рисование и обрабатывает события как компонент, но также содержит кнопку как контейнер.

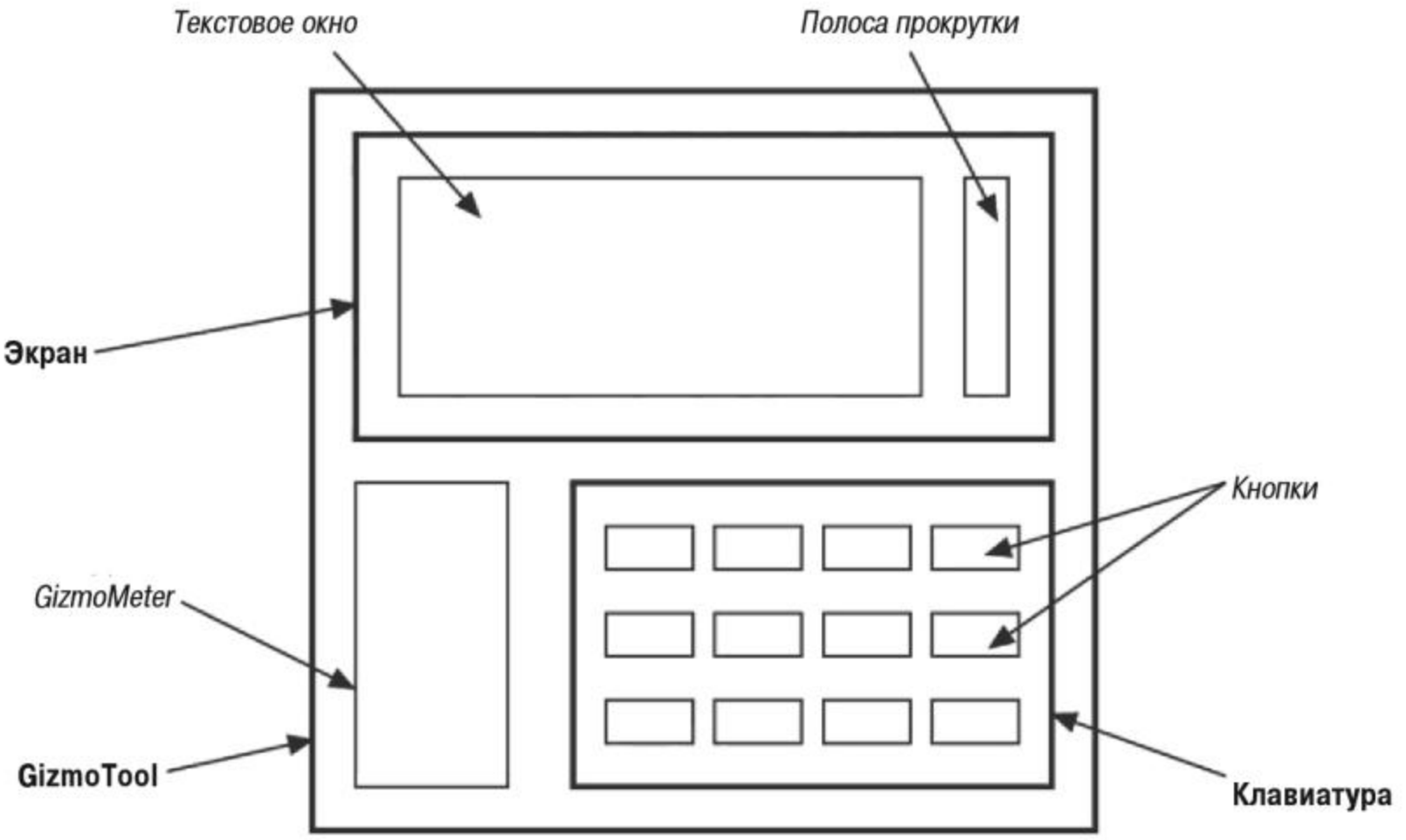

Рис. 2.10. Схема контейнеров (выделены **полужирным**) и компонентов (выделены *курсивом*) Java

Диспетчеры компоновки

После создания объекта `JButton` мы должны поместить его в контейнер, но куда? Объект, называемый `LayoutManager`, определяет местонахождение контейнера `HelloComponent3`, в котором нужно показывать `JButton`. Объект класса `LayoutManager` реализует схему для организации компонентов на экране и настройки их размеров. Существует несколько стандартных диспетчеров компоновки, среди которых можно выбирать, и, конечно, мы можем создавать новые. В нашем случае мы укажем один из стандартных диспетчеров, `FlowLayout`. Окончательным результатом станет то, что кнопка разместится по центру вверху контейнера `HelloComponent3`. У нашего диспетчера компоновки `JFrame` есть еще одна разновидность, которая называется `BorderLayout`. Вы узнаете больше о диспетчерах компоновки в главе 19.

Чтобы добавить кнопку в диспетчер, мы вызываем метод `add()`, который `HelloComponent3` наследует от класса `Container`, передавая объект `JButton` как параметр:

```
add( theButton );
```

`add()` — это метод, унаследованный нашим классом от класса `Container`. Он добавляет наш `JButton` к списку компонентов, которыми

управляет контейнер HelloComponent3. Соответственно, HelloComponent3 ответственен за JButton: он делает кнопку видимой и определяет, где в окне должна размещаться кнопка.

Подклассы и подтипы

Если вы посмотрите на метод add() класса Container, вы увидите, что он использует объект Component в качестве аргумента. В нашем примере мы передали ему объект JButton. Что происходит?

Как мы сказали, JButton является подклассом класса Component. Поскольку подкласс — тип своего надкласса и имеет как минимум те же публичные методы и переменные, Java позволяет нам использовать экземпляр подкласса везде, где мы можем использовать экземпляр своего надкласса. JButton — тип класса Component, поэтому любой метод, который ожидает класс Component в качестве аргумента, примет класс JButton. Обратное, однако, неверно. Сигнатура функции, ожидающая определенный класс, не примет его надкласс в качестве параметра.

Больше событий и интерфейсов

Теперь, когда у нас есть JButton, нам нужно как-то с ним сообщаться — это значит получать события, которые он генерирует. Мы могли просто отслеживать щелчки мыши внутри кнопки и действовать соответственно, но это бы потребовало настройки посредством подклассов JButton и мы бы лишились преимуществ использования заранее предусмотренного компонента. Вместо этого у нас есть приемник объектов HelloComponent3 для событий высокого уровня, соответствующих нажатию на мышь. Объект JButton генерирует особый тип событий, называемый ActionEvent, когда кто-то щелкает по нем мышью. Чтобы получить эти события, мы добавили еще один метод к классу HelloComponent3:

```
public void actionPerformed( ActionEvent e ) {
if ( e.getSource() == theButton )
changeColor();
}
```

Если бы вы повторили предыдущий пример, вы бы не удивились, увидев, что HelloComponent3 теперь объявляет, что он реализует интерфейс ActionListener в дополнение к MouseMotionListener. ActionListener требует, чтобы мы применили метод actionPerformed(), который вызы-

вается всегда, когда возникает ActionEvent. Вы также не удивитесь, увидев, что мы добавили строку к конструктору HelloComponent3, регистрирующему себя (this) как приемник событий действия кнопки:

```
theButton.addActionListener( this );
```

Заметьте, что в этот раз мы регистрируем наш компонент как приемник с другим объектом — кнопкой, в то время как ранее мы запрашивали собственные события.

Метод actionPerformed() заботится обо всех возникающих событиях действий. Во-первых, он убеждается, что источник события (компонент, генерирующий событие) является тем, о чем мы думаем: theButton. Это может показаться излишним, в конце концов, у нас только одна кнопка. Что еще может генерировать событие действия? В этом приложении ничего, но проверить стоит, поскольку другое приложение может иметь много кнопок и вам, возможно, понадобится выяснить, какая из них была нажата. Или вы можете добавить еще одну кнопку к этому приложению позже, и вы не хотите, чтобы из-за этого что-то поломалось. Чтобы это проверить, мы вызываем метод getSource() объекта ActionEvent, e. Затем мы используем операцию ==, чтобы убедиться, что источник события совпадает с theButton.



В Java операция == является проверкой на *тождественность*, а не на эквивалентность; результат истинный, если источник события и theButton являются одним и тем же объектом. Различие между равенством и тождественностью имеет большое значение. Два объекта типа String можно считать равными, если они содержат одни и те же символы, размещенные в одном и том же порядке. При этом они могут представлять собой разные объекты. В главе 7 мы рассмотрим метод equals(), который выполняет проверку на тождественность.

Когда мы установим, что событие e исходит от правой кнопки, мы вызываем наш метод changeColor(), на этом мы заканчиваем.

Вы, возможно, удивитесь, почему нам не нужно менять метод mouseDragged() сейчас, когда у нас есть в приложении объект JButton. Разумное объяснение вот в чем: все, что имеет значение в этом методе, — это координаты события. Мы не особо обеспокоены тем, попадает ли событие в область экрана, занятую другим компонентом. Это означает, что вы можете перемещать текст непосредственно через объект JButton: по-

пробуйте и увидите! В этом случае компоновка контейнеров означает, что кнопка находится наверху нашего компонента, поэтому перетаскиваемый текст под ним.

Замечание относительно цвета

Для управления цветовой схемой мы добавили пару новых переменных и два полезных метода. Мы создаем и инициализируем массив объектов класса `Color`, представляющий цвета, которые мы повторяем, когда нажата кнопка. Мы также объявляем переменную целого типа, которая служит индексом в этом массиве, определяя положение текущего цвета:

```
int colorIndex;
static Color[] someColors = { Color.black, Color.red,
Color.green, Color.blue, Color.magenta };
```

Здесь происходит целый ряд событий. Во-первых, давайте посмотрим на объекты `Color`, которые мы помещаем в массив. Экземпляры `java.awt.Color` представляют цвета; они используются всеми классами пакета `java.awt`, который работает с базовой цветной графикой. Обратите внимание, что мы ссылаемся на такие переменные, как `Color.black` и `Color.red`. Они выглядят как примеры переменных экземпляра объекта, но `Color` не является объектом, это класс. Что все это означает? Далее мы обсудим это.

Статические члены

Класс может содержать переменные и методы, которые делятся между экземплярами класса. Эти делимые методы называются *статическими переменными* и *статическими методами*. Наиболее общее использование статических переменных в классе — хранение предопределенных констант или неизменяемых объектов, которые могут использовать все экземпляры.

У этого подхода два преимущества. Одно из них — это то, что статические значения делятся экземплярами класса; одно и то же значение может быть видно для всех экземпляров. Что более важно, статические члены могут быть доступны, даже если не существует никаких экземпляров класса. В этом примере мы используем статическую переменную `Color.red`, не имея необходимости создавать экземпляр класса `Color`.

Экземпляр класса `Color` представляет видимый цвет. Для удобства класс `Color` содержит некоторые статические, предопределенные объекты с дружелюбными именами `GREEN`, `RED` и `MAGENTA`. Переменная `GREEN`, например, является статическим членом в классе `Color`. Тип данных переменной `GREEN` — `Color`. Внутри, в стране Java, она инициализируется таким образом:

```
public final static Color GREEN = new Color(0, 255, 0);
```

Переменная `GREEN` и другие статические члены класса `Color` не могут изменяться (после того, как они были инициализированы), поэтому они фактически постоянной величины и могут оптимизироваться как таковые виртуальной машиной Java. Альтернативой использованию этих предопределенных цветов является создание цвета вручную, при этом нужно указать красный, зеленый и синий компоненты (RGB), используя конструкторы класса.

Массивы

Далее мы обратим наше внимание на массив. Мы объявили переменную, называемую `someColors`, которая является массивом объектов класса `Color`. В Java массивы являются объектами *первого класса*. Это означает, что сам массив является типом объекта, который знает, как хранить индексированный список каких-то других типов объектов.

Массив индексируется целыми числами; когда вы индексируете массив, результирующим значением является ссылка на объект, который находится в специальном гнезде массива. Наш код использует переменную `colorIndex`, чтобы индексировать `someColors`. Также возможно иметь массив простых примитивных типов, таких как `float`, а не объектов.

Когда мы объявляем массив, мы можем инициализировать его, используя конструкцию с фигурными скобками. Указание разделенного запятой списка элементов внутри фигурных скобок является функцией, которая сообщает компилятору о создании экземпляра массива с этими элементами и приписывает его нашей переменной. Кроме того, мы могли бы просто объявить нашу переменную `someColors`, а затем назначить ему объект типа массив и назначить индивидуальные элементы слотам этого массива. Смотрите главу 5, чтобы получить полное представление о массивах.

Методы для класса `Color`

Теперь у нас есть массив объектов `Color` и переменная, которой можно индексировать массив. Два приватных метода выполняют для нас реальную работу. Модификатор `private` у этих методов указывает, что они могут вызываться только методами в тех же экземплярах класса.

К ним невозможно получить доступ вне объектов, которые их содержат. Мы объявляем, что члены имеют модификатор `private`, чтобы спрятать подробные внутренние механизмы класса от внешнего мира. Это называется *инкапсуляция* и является еще одним основным положением объектно-ориентированной разработки, а также хорошей практикой программирования. Приватные методы созданы как вспомогательные функции для использования исключительно в реализации класса.

Первый метод — `currentColor()` — является просто удобной программой, которая возвращает объект класса `Color`, представляющий текущий цвет текста. Он возвращает объект класса `Color` в массиве `someColors` в индексе, указанном нашей переменной `colorIndex`:

```
synchronized private Color currentColor() {
return someColors[colorIndex];
}
```

Мы точно так же могли бы использовать выражение `someColors [colorIndex]` везде, где мы используем `currentColor()`; однако создание методов для скрытия общих задач является еще одним способом укрыться от деталей нашего класса. В альтернативной реализации мы, возможно, отправили бы детали всего кода, связанного с цветом, в отдельный класс. Мы бы могли создать класс, который принимает массив цветов в свой конструктор и затем предоставляет два метода: один для запроса текущего цвета и один для перехода к другому цвету (просто пища для размышления).

Второй метод — `changeColor()` — отвечает за инкрементирование переменной `colorIndex` для указания на следующий класс `Color` в массиве. `changeColor()` вызывается нашим методом `actionPerformed()` каждый раз, когда нажимается кнопка:

```
synchronized private void changeColor() {
// Изменить индекс на следующий цвет, затруднительно
```

```
if ( ++colorIndex == someColors.length )
colorIndex = 0;
setForeground( currentColor() ); // Использовать новый цвет.
repaint();
}
```

Здесь мы увеличиваем переменную `colorIndex` и сравниваем ее с длиной массива `someColors`. Все объекты массива имеют переменную, называющуюся `length`, которая указывает на количество элементов в массиве. Если мы достигнем конца массива, мы совершим циклический переход к его началу, сбросив индекс до ноля. Мы пометили это комментарием, чтобы показать, что мы делаем нечто сомнительное. Но мы вернемся к этому очень скоро. После смены только что выбранного цвета, мы делаем две вещи. Во-первых, вызываем метод `setForeground()`, который меняет цвет, используемый для рисования текста в нашем компоненте. Затем вызываем `repaint()`, чтобы заставить компонент перерисовать перетаскиваемое сообщение новым цветом. Что за ключевое слово `synchronized`, которое появляется перед нашими методами `currentColor()` и `changeColor()`? При синхронизации приходится иметь дело с потоками, которые мы исследуем в следующем разделе. Все, что вам нужно знать сейчас: ключевое слово `synchronized` указывает на то, что эти два метода никогда не могут работать одновременно. Они всегда должны работать взаимоисключающим образом.

Причина этого связана с сомнительным образом, которым мы инкрементируем наш индекс. Обратите внимание на то, что в `changeColor()` мы инкрементируем `colorIndex` перед проверкой его значения. Прямо говоря, это означает, что в течение некоего короткого промежутка времени, пока Java прогоняет наш код, `colorIndex` может иметь значение, которое находится за концом нашего массива.

Если бы в этот же момент случайно запустился наш метод `currentColor()`, мы бы увидели ошибку при выполнении array out of bounds («массив выходит за пределы»).

Теперь нам будет легче решить эту проблему при помощи простых арифметических действий перед сменой значения, но этот простой пример демонстрирует более общие проблемы синхронизации, которые нам нужно решать. Мы используем этот пример, чтобы проиллюстрировать применение ключевого слова `synchronized`. В следующем разделе вы увидите, что Java справляется с этими проблемами сравнительно просто благодаря поддержке синхронизации на уровне языка.

HelloJava4: месть Netscape

Мы исследовали несколько свойств Java при помощи трех первых версий приложения HelloJava. Но до нынешнего момента наше приложение было довольно пассивным; оно было полностью *событийно-управляемым*, терпеливо ожидающим событий и отвечающим прихотям пользователя. Теперь наше приложение начнет проявлять кое-какую инициативу. HelloJava4 будет мерцать*! Ниже представлен код:

```
//файл: HelloJava4.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HelloJava4
{
public static void main( String[] args ) {
JFrame frame = new JFrame( "HelloJava4" );
frame.add( new HelloComponent4("Hello, Java!") );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setSize( 300, 300 );
frame.setVisible( true );
}
}
class HelloComponent4 extends JComponent
implements MouseMotionListener, ActionListener, Runnable
{
String theMessage;
int messageX = 125, messageY = 95; // Координаты сообщения
JButton theButton;
int colorIndex; // Текущий индекс в переменной someColors.
static Color[] someColors = {
Color.black, Color.red, Color.green, Color.blue, Color.magenta };
boolean blinkState;
public HelloComponent4( String message ) {
theMessage = message;
theButton = new JButton("Change Color");
setLayout( new FlowLayout() );
add( theButton );
theButton.addActionListener( this );
```

* Заголовок раздела «Месть Netscape» относится к малоизвестному html-тегу <BLINK>, использовавшемуся в ранних версиях веб-браузера Netscape.

```
addMouseMotionListener( this );
Thread t = new Thread( this );
t.start();
}
public void paintComponent( Graphics g ) {
g.setColor(blinkState ? getBackground() : currentColor());
g.drawString(theMessage, messageX, messageY);
}
public void mouseDragged(MouseEvent e) {
messageX = e.getX();
messageY = e.getY();
repaint();
}
public void mouseMoved(MouseEvent e) { }
public void actionPerformed( ActionEvent e ) {
if ( e.getSource() == theButton )
changeColor();
}
synchronized private void changeColor() {
if (++colorIndex == someColors.length)
colorIndex = 0;
setForeground( currentColor() );
repaint();
}
synchronized private Color currentColor() {
return someColors[colorIndex];
}
public void run() {
try {
while(true) {
blinkState = !blinkState; // Переключить blinkState.
repaint(); // Показать изменения.
Thread.sleep(300);
}
} catch (InterruptedException ie) { }
}
}
```

Скомпилируйте и запустите эту версию приложения `HelloJava`, как и предыдущие. Вы увидите, что текст действительно мерцает. Наши извинения, если вас это раздражает, все это ради обучения.

Потоки выполнения

Все изменения, которые мы внесли в `HelloJava4`, относятся к установлению отдельного потока выполнения, чтобы заставить текст мерцать. Java является *многопоточным* языком, и это означает, что может быть много путей выполнения эффективного прогона одновременно. *Поток выполнения* — это алгоритм управления внутри программы. По существу потоки идентичны процессам. В отличие от процессов многочисленные потоки делят одно и то же пространство программы — это значит, что они могут делить переменные и методы (но также иметь свои собственные локальные переменные). Потоки являются достаточно легковесными в сравнении с процессами, поэтому одно приложение может работать сразу с множеством потоков (возможно, сотнями или тысячами) параллельно.

Многопоточность дает приложению возможность управляться с большим количеством разнообразных задач одновременно. Легко представить себе множество вещей, происходящих одновременно в таком приложении, как веб-браузер. Пользователь может слушать аудиозапись во время прокрутки изображения; в то же время браузер может загружать другое изображение. Многопоточность особенно полезна в приложениях, основанных на графическом интерфейсе пользователя, потому что она улучшает интерактивное выполнение этих приложений.

К сожалению, для нас программирование с множеством потоков может быть довольно проблематичным. Сложность заключается в следующем: необходимо удостовериться, что рутинные методы реализованы таким образом, чтобы они могли запускаться параллельно более чем одним потоком за один раз. Если рутинный метод изменяет значение множества параметров состояния, например, может быть важным, что эти изменения происходят одновременно без наложения изменений, влияющих друг на друга. Далее в этом разделе мы кратко исследуем проблему координирования доступа множественных потоков к общим данным. В других языках синхронизация потоков может быть чрезвычайно сложной и подверженной ошибкам. Вы увидите, что Java дает вам мощные инструменты, которые помогают решать множество этих проблем. Мы подробно рассмотрим потоки в главе 9.

Система выполнения Java создает множество потоков и управляет ими. (Как именно — зависит от реализации.) Мы уже упомянули поток перекрашивания, управляющий запросами `repaint()` и обработкой событий компонентов графического интерфейса пользователя, которые

относятся к пакетам `java.awt` и `javax.swing`. Наши примеры приложений осуществили большую часть своей работы одним потоком. Такие методы, как `mouseDragged()` и `actionPerformed()`, вызываются окном потока и запускаются в свою очередь его потоком. Аналогично наш конструктор `HelloComponent` работает как часть потока главного приложения (метод `main()`). Если бы мы, например, рассматривали бесконечный цикл в нашем конструкторе, наше приложение бы никогда не появилось, поскольку оно бы никогда не завершило инициализацию. Если бы мы хотели, чтобы приложение выполняло какой-то большой объем работы, такой как анимация, длинное вычисление или коммуникация, мы бы создали отдельные потоки для этих задач.

Класс `Thread`

Как вы, должно быть, догадались, потоки создаются и контролируются, как объекты класса `Thread`. Экземпляр класса `java.lang.Thread` соответствует одному потоку. Он содержит методы для начала, управления и прерывания выполнения потока. Сейчас мы планируем создать объект класса `Thread`, чтобы управлять нашим мигающим текстом. Мы вызываем метод `start()` класса `Thread` для начала выполнения. Когда поток запускается, он продолжает работать, пока не выполнит свою работу, мы прерываем его или останавливаем приложение.

Итак, как мы сообщаем потоку, какой метод использовать? Что ж, объект класса `Thread` довольно разборчив; он всегда ожидает выполнения метода, называемого `run()` для выполнения действия потока. Метод `run()`, однако, может после некоторого воздействия располагаться в любом классе, который мы захотим.

Мы указываем положение метода `run()` одним из двух способов. Во-первых, сам класс `Thread` имеет метод, называемый `run()`. Один из способов выполнения некоего кода Java — это создать подкласс класса `Thread` и перезаписать его метод `run()`, чтобы сделать нашу компоновку. Вызов метода `start()` подкласса объекта приведет к тому, что метод `run()` будет выполняться в отдельном потоке.

Обычно не очень желательно создавать подкласс класса `Thread`, который будет содержать наш метод `run()`. У класса `Thread` есть конструктор, который берет объект в качестве аргумента. Если мы создадим объект класса `Thread`, используя этот конструктор, и вызовем его метод `start()`, класс `Thread` выполнит метод `run()` объекта-аргумента, а не свой. Для

выполнения этого Java нуждается в гарантии того, что объект, который мы передаем ему, действительно содержит совместимый метод `run()`. Мы уже знаем, как создать такую гарантию: мы используем интерфейс. Java предоставляет интерфейс, называющийся `Runnable`, который может реализовываться любым классом, который хочет стать классом `Thread`.

Интерфейс `Runnable`

Мы реализовали интерфейс `Runnable` в `HelloComponent4`. Чтобы создать поток, объект `HelloComponent4` передает себя (`this`) конструктору `Thread`. Это означает, что `HelloComponent4` должен реализовывать интерфейс `Runnable` реализацией метода `run()`. Этот метод вызывается автоматически, когда система выполнения нуждается в запуске потока.

Мы указываем, что класс реализует интерфейс в нашем объявлении класса:

```
public class HelloComponent4
extends JComponent
implements MouseMotionListener, ActionListener, Runnable {...}
```

Во время компиляции компилятор Java проверяет, что мы действительно придерживаемся этой инструкции. Мы завершили работу, добавив соответствующий метод `run()` в `HelloComponent4`. Он не берет аргументов и не возвращает никаких значений. Наш метод `run()` выполняет мерцание, изменяя цвет нашего текста несколько раз за секунду. Это очень короткая рутинная операция, но мы собираемся отложить ее изучение, пока не рассмотрим до конца тему о классе `Thread`.

Запуск потока

Мы хотим, чтобы мерцание начиналось при запуске приложения, поэтому мы создадим поток, инициализировав код в конструкторе `HelloComponent4`. Это займет всего две строки:

```
Thread t = new Thread(this);
t.start();
```

Во-первых, конструктор создает новый экземпляр класса `Thread`, передавая ему объект, который содержит метод `run()`, применяемый к конструктору. Поскольку сам `HelloComponent4` содержит метод `run()`, мы

передаем особую переменную this конструктору. Переменная this всегда ссылается на наш объект. После создания нового класса Thread, мы вызываем его метод start(), чтобы начать выполнение. Это в свою очередь вызывает метод run() компонента HelloComponent4 в новом потоке.

Выполнение кода в классе Thread

Наш метод run() делает свою работу, устанавливая значение переменной blinkState. Мы добавили blinkState, булеву переменную, которая может принимать значения true или false, чтобы показать, загораемся мы сейчас или меркнем:

```
boolean blinkState;
```

Вызов setColor() был добавлен к нашему методу paintComponent() для управления мерцанием. Когда blinkState имеет значение true, вызов setColor() рисует текст цвета фона, делая его невидимым.

```
g.setColor(blinkState ? getBackground() :
currentColor());
```

Здесь мы немногословны и используем тернарную операцию в стиле языка C, чтобы вернуть одно или два альтернативных значения цвета, основанных на значении blinkState. Если blinkState имеет значение true, значением является то, которое возвращается методом getBackground(). Если это значение false, значением является то, которое возвращается методом currentColor().

В конце концов, мы подходим к самому методу run():

```
public void run() {
try {
while( true ) {
blinkState = !blinkState;
repaint();
Thread.sleep(300);
}
} catch (InterruptedException ie) {}
}
```

По сути run() является бесконечным циклом while — это означает, что цикл отрабатывает непрерывно, пока поток не прерывается выходом

из приложения (по сути, не очень хорошая идея, но работает для этого примера).

Тело цикла выполняет три операции во время каждого прохода:

- перебрасывает значение `blinkState` на противоположное, используя операцию `not` (`!`);
- вызывает `repaint()`, чтобы перерисовать текст;
- засыпает на 300 миллисекунд (около трети секунды).

`sleep()` является статическим методом класса `Thread`. Метод может вызываться отовсюду и имеет эффект погружения выполняемого в данный момент потока в спячку на определенное количество миллисекунд. В результате мы имеем приблизительно три мерцания в секунду. Концепция `try/catch`, описанная в следующем разделе, отлавливает все ошибки в вызове метода `sleep()` класса `Thread` и в этом случае игнорирует их.

Исключения

Оператор `try/catch` в Java хранит специальные условия, называемые *исключениями*. Исключение — это сообщение, обычно посылаемое в ответ на ошибку во время выполнения оператора или метода. Когда возникает исключающее условие, создается объект, который содержит информацию об отдельно взятой проблеме или условии. Исключения похожи на события. Java останавливает выполнение в том месте, где возникает исключение, и говорится, что исключение *выбрасывается* этой частью кода. Как и событие, исключение должно куда-то передаваться и храниться. Говорится, что часть кода, которая получает объект-исключение, *захватывает* исключение. Исключение вызывает резкую остановку выполнения инициирующей части кода и передает управление коду, который получает объект-исключение.

Конструкция `try/catch` позволяет вам отлавливать исключения для части кода. Если исключение вызвано инструкцией внутри оператора `try`, Java пытается передать исключение соответствующему оператору `catch`. Оператор `catch` выглядит как объявление метода с одним аргументом и без типа возврата функции.

```
try {
...
```

```
} catch ( SomeExceptionType e ) {
...
}
```

Если Java находит оператор `catch` с типом аргумента, который соответствует типу исключения, вызывается этот оператор `catch`. Оператор `try` может иметь множество операторов `catch` с разными типами аргументов; Java выбирает подходящий путем, аналогичным выбору перегруженных методов. Вы можете отловить множество типов исключений из блока кода. В зависимости от подброшенного типа исключения выполняется подходящий оператор `catch`.

Если нет никакого оператора `try/catch`, окружающего код, или не находится подходящий оператор `catch`, исключение подбрасывается вызываемому методу. Если исключение не отлавливается на этом этапе, оно передается на другой уровень и так далее, пока исключение не будет обработано или виртуальная машина Java не выведет ошибку и не выйдет. Это дает очень гибкий механизм обработки ошибок, так что исключения в глубоко расположенных вызовах могут всплыть на поверхность стека вызовов для обработки. Как программисту, вам необходимо знать, какие исключения может генерировать отдельный оператор. По этой причине в Java необходимо, чтобы методы объявляли исключения, которые они могут выбросить. Если метод не обрабатывает само исключение, он должен указать, что может перебросить исключение, и его вызывающий метод будет знать, что ему, возможно, придется обращаться с ним. См. главу 4 для полного изучения исключений и операторов `try/catch`.

Зачем нам нужен оператор `try/catch` в методе `run()`? Какой тип исключения может подкинуть метод `sleep()` класса `Thread` и почему мы заботимся об этом, если мы, кажется, не проверяем исключения больше нигде? При определенных обстоятельствах метод `sleep()` класса `Thread` может выбрасывать исключение `InterruptedException`, указывая на то, что он был прерван другим потоком.

Поскольку метод `run()`, указанный в интерфейсе `Runnable`, не объявляет, что он может выбросить исключение `InterruptedException`, мы должны отловить его самостоятельно, иначе компилятор будет «выражать недовольство». Оператор `try/catch` в нашем примере имеет пустое условие `catch`, которое означает, что он обрабатывает исключение, игнорируя его. В этом случае функционал нашего потока так прост, что не имеет значения, прерывается он или нет (и он не будет этого делать

ни в коем случае). Все другие методы, которые мы использовали, либо обрабатывают свои собственные исключения, либо выбрасывают только универсальные исключения, называемые `RuntimeExceptions`, которые предположительно возможны везде и не должны явно объявляться.

Синхронизация

В любое время в приложении может быть запущено множество потоков. Если мы явно не координируем их, эти потоки будут выполнять методы, не обращая внимания на то, что выполняют другие потоки. Проблемы могут возникать, когда эти методы работают с одними и теми же данными. Если один метод меняет значение некоторых переменных, в это же время другой метод читает эти переменные, возможно, что читающий поток может отлавливать их посреди операции и получать некоторые переменные со старыми значениями и некоторые — с новыми. В некоторых приложениях эта ситуация может вызывать критические ошибки.

В наших примерах `HelloJava` оба наших метода — `paintComponent()` и `mouseDragged()` — имеют доступ к переменным `messageX` и `messageY`.

Не зная ничего больше о реализации среды Java, мы должны предположить, что эти методы могут вызываться разными потоками и запускаться параллельно. `paintComponent()` может осуществляться в то время, как `mouseDragged()` обновляет `messageX` и `messageY`. На этом этапе данные находятся в неопределенном состоянии и, если методу `paintComponent()` повезет, он может получить новые значения `x` со старыми значениями `y`.

К счастью, Swing не позволит этому произойти в данном случае, поскольку вся активность события управляется одним потоком, и мы в любом случае вряд ли заметили, если бы такое случилось в этом приложении. Однако в наших методах `changeColor()` и `currentColor()`мы видели другой случай, который является представителем более серьезной ошибки «вне границ».

Модификатор `synchronized` говорит Java о получении *замка* для объекта, который содержит метод перед выполнением этого метода. Только один метод объекта может быть заблокирован в любой момент времени, то есть только один синхронизированный метод в данном объекте может работать в это время. Это позволяет методу менять данные и оставлять их в согласованном состоянии прежде, чем параллельно работающему

методу будет дозволено получить доступ к нему. Когда метод выполнен, он снимает блокировку с класса.

В отличие от синхронизации в других языках ключевое слово `synchronized` в Java обеспечивает блокировку на уровне языка. Это означает, что ни в коем случае вы не забудете разблокировать класс. Даже если метод выдает исключение или поток остановлен, Java снимет блокировку. Это свойство делает программирование при помощи потоков в Java более простым, чем в других языках. См. главу 9 для получения подробностей по координации потоков и общих данных.

Что ж, пришла пора попрощаться с `HelloJava`. Мы надеемся, что вы поняли большинство свойств языка Java и это поможет вам в детальном изучении программирования на Java. Если вы немного озадачены материалом, изложенным здесь, не унывайте. В книге мы будем возвращаться ко всем изложенным здесь важным темам в разных главах.

Глава 3
ИНСТРУМЕНТЫ ДЛЯ РАБОТЫ

Хотя вы наверняка будете писать большую часть своего кода на Java в таких средах IDE, как Eclipse, NetBeans или любимая IDE автора — Intellij IDEA, все основные инструменты, необходимые вам для создания приложения Java, включены в набор для разработки Java-приложений (JDK), который вы, скорее всего, скачали с сайта Oracle для версии 7. В этом разделе мы обсудим некоторые из этих консольных инструментов, которые вы можете использовать для компиляции, запуска и архивирования приложений Java. В JDK включено много дополнительных инструментов разработчика, которые мы еще рассмотрим в этой книге.

Для введения в среду IDE Eclipse и просмотра инструкций для загрузки всех примеров в этой книге, таких как проект Eclipse, см. приложение A. В главе 22 мы представим IDE NetBeans и обсудим архитектуру компонентов JavaBeans, так что вы получите дополнительный опыт в среде разработки графического интерфейса пользователя.

Среда JDK

После установки Java 7, главная команда выполнения `java` может быть готова к выполнению автоматически. Однако многие другие команды, предоставляемые с JDK, могут не быть доступны, пока вы не добавите папку Java *bin* в вашу выполняемую ветвь. Следующие команды показывают, как сделать это в операционной системе OS X и Windows. Вам, конечно, придется поменять путь, чтобы он соответствовал версии Java, которую вы установили.

```
# Mac OS X
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_09.jdk/
Contents/Home
export PATH=$PATH:$JAVA_HOME/bin
# Windows
```

```
set JAVA_HOME=c:\Program Files\Java\jdk1.7.0_09
set PATH=%PATH%;%JAVA_HOME%\bin
```

В операционной системе OS X ситуация может быть более сложной, поскольку последние версии поставляются с «заглушками» для инсталлированных команд Java. Если вы попытаетесь запустить одну из этих программ, OS попросит вас скачать Java. На момент написания этой книги операционная система OS X до сих пор поставляется с версией Java 6, поэтому придется скачать версию 7 и установить свою среду, как показано.

Если есть сомнения, вашим тестом `go-to` для определения того, какой версией инструментов вы пользуетесь, будет использование флага `-version` в командах `java` и `javac`.

```
java -version
# java version "1.7.0_07"
# Java(TM) SE Runtime Environment (build 1.7.0_07-b10)
# Java HotSpot(TM) 64-Bit Server VM (build 23.3-b01, mixed mode)
javac -version
# javac 1.7.0_07
```

Виртуальная машина Java

Виртуальная машина Java — это программное обеспечение, которое реализует систему выполнения Java и выполняет приложения Java. Это может быть отдельное приложение, как команда `java`, которая поступает с JDK, или встроенное в более крупное приложение, как веб-браузер. Обычно сам интерпретатор является родным приложением, поставляемым с каждой платформой, которая затем инициализирует другие инструменты, написанные на языке Java. Такие инструменты, как компиляторы Java и IDE, часто реализуются прямо в Java, чтобы максимизировать их портативность и расширяемость. Например, NetBeans является приложением, написанным полностью на Java.

Виртуальная машина Java выполняет все операции прогона Java. Она загружает файлы классов Java, проверяет классы из недостоверных источников и выполняет скомпилированный байт-код. Она управляет памятью и ресурсами системы. Хорошие реализации также производят динамическую оптимизацию, компилируя байт-код Java в родные машинные команды.

Запуск приложения Java

Отдельное приложение Java должно иметь хотя бы один класс, содержащий метод, называемый `main()`, который является первым кодом для выполнения при запуске. Чтобы запустить приложение, запустите виртуальную машину, уточняя, что класс является аргументом. Вы можете уточнить опции интерпретатору, а также аргументы, передаваемые приложению:

```
% java [опции интерпретатора] имя_класса [аргументы приложения]
```

Класс должен быть указан как полностью квалифицированное имя класса, включая имя пакета, если оно есть. Отметьте, однако, что вы не включаете расширение файла *.class*. Вот пара примеров:

```
%java animals.birds.BigBird
%java MyTest
```

Интерпретатор ищет класс в *пути к классам*, списке директорий и файлов архива, где хранятся классы. Мы детально обсудим путь к классам в следующем разделе. Путь к классам может быть определен *переменной среды* или консольной опцией `-classpath`. Если присутствуют оба, используется консольная опция.

С другой стороны, команда `java` может использоваться для запуска выполняемого файла архива Java (JAR):

```
% java -jar spaceblaster.jar
```

В этом случае файл JAR включает метаданные с именем класса запуска, включающие метод `main()`, и путь к классам сам становится файлом JAR.

После загрузки первого класса и выполнения метода `main()`, приложение может ссылаться на другие классы, начинать дополнительный поток и создавать свой интерфейс пользователя или другие структуры, показанные на рис. 3.1.

Метод `main()` должен иметь правильную *подпись метода*. Подпись метода является набором информации, которая определяет метод. Она включает имя метода, аргументы и тип возврата функции, а также тип и модификаторы видимости. Метод `main()` должен быть методом `public`, `static`, который принимает массив объектов класса `String` в качестве своего аргумента и не возвращает никакого значения (`void`):

```
public static void main ( String [] myArgs )
```

Тот факт, что `main()` является методом `public` и `static`, просто означает, что он глобально доступный и что он не может вызываться непосредственно по имени. Мы обсудим сущность таких модификаторов видимости, как `public` и смысл `static`, в главах с 4 по 6.

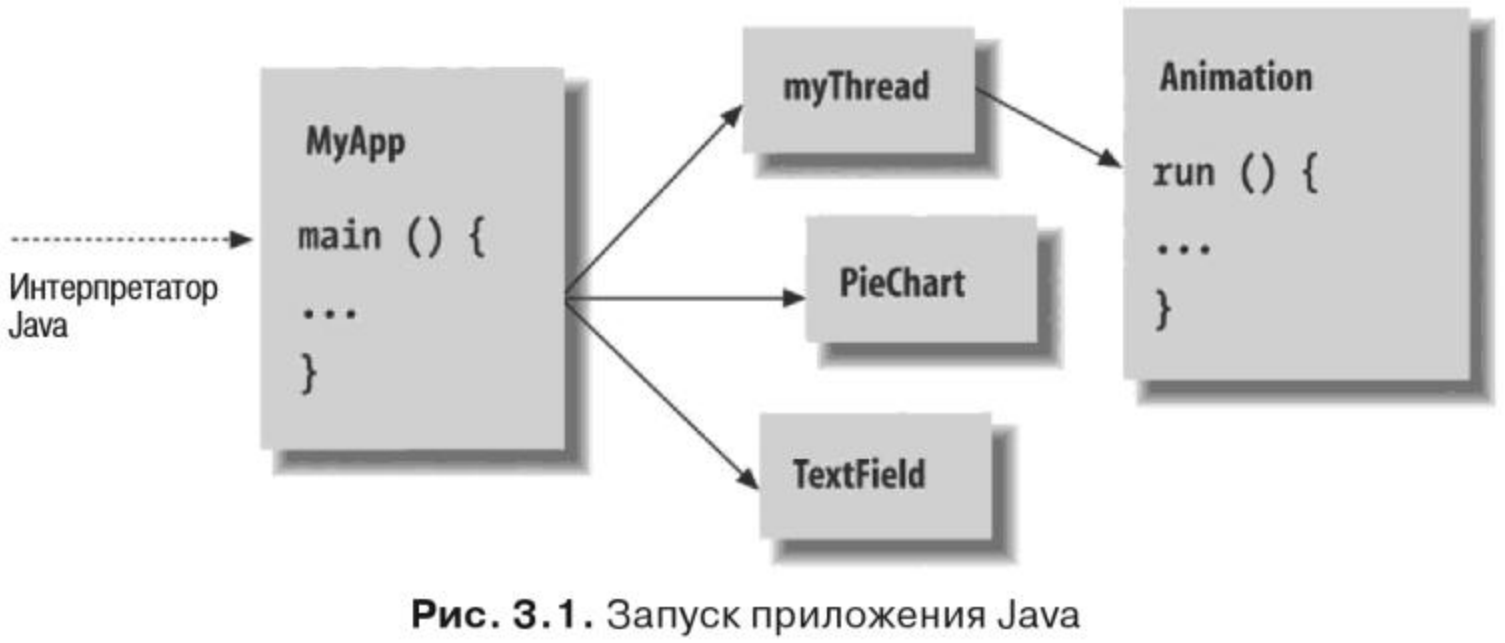

Рис. 3.1. Запуск приложения Java

Единственный аргумент метода `main()`, массив объектов класса `String`, содержит консольные аргументы, передаваемые приложению. Имя параметра не имеет значения; важен только тип. В Java содержание переменной `myArgs` является массивом. В Java массивы знают, как много элементов они содержат, и с радостью могут предоставить эту информацию.

```
int numArgs = myArgs.length;
```

`myArgs[0]` является консольным аргументом и т. д.

Интерпретатор Java продолжает работать, пока метод `main()` начального класса файла не вернется и пока все потоки, которые он начал, не выйдут. Специальные потоки, разработанные как потоковые *демоны*, автоматически прерываются, когда выполнена остальная часть приложения.

Свойства системы

Хотя можно считывать переменные среды выполнения непосредственно из среды Java, это не приветствуется конфигурацией приложения. Наоборот, Java позволяет любому количеству значений *свойств системы* передаваться приложению, когда запускается виртуальная машина. Свойства системы являются просто парой строк имя-значение, которые доступны приложению через статический метод `System`.

getProperty(). Вы можете использовать эти свойства как более структурированную и портативную альтернативу консольным аргументам и переменным среды для предоставления общей информации о конфигурации вашему приложению при запуске. Каждое свойство системы передается интерпретатору в командной строке с использованием опции -D, за которой следует name=value. Например:

```
% java -Dstreet=sesame -Dscene=alley animals.birds.BigBird
```

Значение свойства street затем доступно таким способом:

```
String street = System.getProperty("street");
```

Приложение может получить свою конфигурацию бесчисленным множеством способов, например, посредством файлов и сетевой конфигурации во время выполнения.

Переменная среды Classpath

Понятие *пути* должно быть знакомо всем, кто работал на платформах DOS или Unix. Это переменная среды, которая предоставляет приложению список мест, где искать ресурсы. Наиболее распространенным примером является путь к выполняемым программам. В командной оболочке Unix переменная среды PATH является упорядоченным списком, разделенным двоеточиями директорий, которые ищутся по порядку, когда пользователь набирает имя команды. Переменная среды Java CLASSPATH — просто список положений, которые разыскиваются файлами класса Java. Интерпретатор Java, как и компилятор, использует CLASSPATH при поиске пакетов и классов Java.

Элемент пути классов может быть директорией или файлом JAR. Java также поддерживает архивы в общепринятом формате ZIP, но JAR и ZIP в действительности являются одним и тем же форматом. JAR — это просто архивы, которые включают дополнительные файлы (метаданные), описывающие содержание каждого архива. Файлы JAR создаются при помощи утилиты jar; многие инструменты для создания архивов ZIP являются публично доступными и могут также быть использованы для просмотра и создания файлов JAR. Формат архива позволяет передачу большой группы классов и их ресурсов в одном файле; во время выполнения Java автоматически извлекает индивидуальные файлы классов из архивов при необходимости.

Определенные средства и формат установки пути классов варьируется от системы к системе. В системе Unix (включая OS X) вы устанавливаете переменную среды `CLASSPATH` со списком директорий, разделенным двоеточием и файлами архивов класса:

```
%export CLASSPATH=/home/vicky/Java/classes:/home/josh/lib/foo.jar:
```

Этот пример устанавливает путь класса с тремя местонахождениями: директория с главной страницей пользователя, файл JAR в директории другого пользователя и текущая директория, которая всегда указывается точкой (.). Последний компонент пути класса, текущая директория, полезен, когда вы возитесь с классами.

В системе Windows переменная среды `CLASSPATH` устанавливается списком директорий, разделенным точкой с запятой и файлами архива классов:

```
C:\> set CLASSPATH=C:\home\vicky\Java\classes;C:\home\josh\lib\foo.jar;
```

Программа запуска Java и другие консольные инструменты знают, как найти внутренние классы, которые являются классами, входящими в каждую установку Java. Классы в пакетах `java.lang`, `java.io`, `java.net` и `javax.swing`, например, все являются внутренними классами, поэтому вам не нужно включать их в ваш путь класса.

Путь класса также может включать специальный символ *, который покрывает все файлы JAR в директории. Например:

```
export CLASSPATH=/home/pat/libs/*
```

Чтобы найти другие классы, интерпретатор Java ищет элементы пути классов по порядку. Поиск объединяет положение пути и компоненты полностью уточненного имени класса. Например, рассмотрим класс `animals.birds.BigBird`. Поиск директории пути класса */usr/lib/java* означает, что интерпретатор ищет индивидуальные файлы класса в */usr/lib/java/animals/birds/BigBird.class*. Поиск архива ZIP или JAR, например, в пути класса */home/vicky/myutils.jar* означает, что интерпретатор ищет файл компонентов *animals/birds/BigBird.class* в этом архиве.

Для системы выполнения Java `java`, и компилятора Java, `javac`, класс пути может быть также установлен опцией `-classpath`:

```
% javac -classpath /home/pat/classes:/utils/utils.jar:. Foo.java
```

Если вы не установите переменную среды `CLASSPATH` или консольную опцию, путь класса присвоится текущей директории (`.`); это значит, что файлы в вашей текущей директории обычно доступны. Если вы измените путь класса и не включите текущую директорию, эти файлы перестанут быть доступными.

Мы подозреваем, что около 80% проблем, с которыми сталкиваются новички, когда изучают Java, связаны с путем класса. Возможно, вы захотите обратить особое внимание на установку и поверку пути класса в начале работы. Если вы работаете в IDE, это может избавить вас от некоторых или всех проблем управления путем класса. В итоге, однако, понимание пути класса и точное знание о том, что находится внутри него, когда приложение работает, очень важно для здоровья вашего рассудка в долгосрочной перспективе. Команда `javap`, которая будет обсуждаться далее, может быть полезна в отладке проблем пути класса.

Утилита `javap`

Полезным инструментом, о котором нужно знать, является утилита `javap`. С `javap` вы можете напечатать описание компилированного класса. Вам не нужен исходный код, вам даже не нужно знать, где именно он находится, только то, что он в пути класса. Например:

```
% javap java.util.Stack
```

выводит информацию о классе `java.util.Stack`:

```
Скомпилировано из "Stack.java"
public class java.util.Stack<E> extends java.util.Vector<E> {
public java.util.Stack();
public E push(E);
public synchronized E pop();
public synchronized E peek();
public boolean empty();
public synchronized int search(java.lang.Object);
}
```

Это очень полезно, если у вас нет под рукой другой документации, и также может помочь в отладке проблем пути класса. Используя `javap`, вы можете определить, находится ли класс в пути класса, а иногда даже версию, на которую вы смотрите (многие проблемы пути класса вклю-

чают дублирование классов в пути класса). Если вам действительно любопытно, попробуйте `javap` при помощи опции `-c`, благодаря чему вы также можете вывести инструкции виртуальной машины Java для каждого метода класса!

Компилятор Java

В этом разделе мы скажем пару слов о `javac`, компиляторе Java в JDK. Компилятор `javac` полностью написан на Java, поэтому доступен для многих платформ, которые поддерживают систему выполнения Java. `Javac` превращает исходный код Java в компилированный класс, который содержит байт-код Java. По обыкновению исходные файлы имеют названия с расширением *.java*. Получающиеся классовые файлы имеют расширение `.class`. Каждый файл исходного кода считается одной единицей компиляции. Как вы увидите в главе 6, классы в определенной единице компиляции имеют определенные общие черты, такие как инструкции `package` и `import`.

`Javac` разрешает один публичный класс на файл и настаивает на том, чтобы файл имел то же имя, что и класс. Если имя файла и класса не совпадают, `javac` выдает ошибку компиляции. Один файл может содержать множество классов, пока один из классов публичный и называется как файл. Избегайте упаковки множества классов в один исходный файл. Упаковка классов вместе в файл *.java* только поверхностно связывает их. В главе 6 мы поговорим о внутренних классах, классах, которые содержат другие классы и интерфейсы.

Для примера поместите следующий исходный код в файл *BigBird.java*:

```
package animals.birds;
public class BigBird extends Bird {
...
}
```

Далее скомпилируем его при помощи:

```
% javac BigBird.java
```

В отличие от интерпретатора Java, который берет в качестве аргумента только имя класса, `javac` нуждается в имени файла (с расширением *.java*) для работы. Предыдущая команда создает класс *BigBird.class*

в той же директории, что и исходный файл. Хотя для этого примера хорошо видеть файл класса в той же директории, что и источник, для большинства реальных приложений вам необходимо хранить файл класса в определенном месте в пути класса.

Вы можете использовать опцию `-d` с `javac`, чтобы указать альтернативную директорию для хранения файлов класса, которые генерирует `javac`. Указанная директория используется как корень класса иерархически, поэтому файлы *.class* помещаются в эту папку или ее подпапку в зависимости от того, содержится ли класс в пакете. (Компилятор создает промежуточные подпапки автоматически, если необходимо.) Например, мы можем использовать следующую команду для создания файла *BigBird.class* в расположении */home/vicky/Java/classes/animals/birds/BigBird.class:*

```
% javac -d /home/vicky/Java/classes BigBird.java
```

Вы можете указать много файлов .java одной командой javac; компилятор создает файл класса для каждого исходного файла. Но вам не нужно перечислять другие классы, на которые ссылается ваш, поскольку они в пути класса в исходной либо скомпилированной форме. Во время компиляции Java разрешает все другие библиотеки классов, используя путь класса.

Компилятор Java ведет себя более разумно, чем ваш среднестатистический компилятор, замещая некоторый функционал утилиты `make`. Например, `javac` сравнивает время изменения источника и файлов класса для всех классов и перекомпилирует их по необходимости. Скомпилированный класс Java помнит исходный файл, из которого он был скомпилирован, и до тех пор, пока исходный файл доступен, `javac` может перекомпилировать его, если необходимо. Если в предыдущем примере класс `BigBird` ссылается на другой класс, `animals.furry.Grover`, `javac` ищет исходный файл *Groverjava в* пакете `animals.furry` и перекомпилирует его, если это необходимо, чтобы сделать `Grover.class` актуальным.

Однако по умолчанию `javac` проверяет только исходные файлы, на которые ведут ссылки непосредственно с других исходных файлов. Это означает, что если у вас есть устаревший файл класса, на который ссылается только актуальный файл класса, он может быть незамечен и перекомпилирован. По этой и многим другим причинам в проектах нередко используется настоящая утилита построения, такая как `Ant` в `Apache` для

управления построениями, упаковкой и другим. Мы обсудим `Ant` в главе 16. Наконец важно отметить, что `javac` может компилировать приложение, даже если существуют только скомпилированные (бинарные) версии некоторых доступных классов. Вам не нужен исходный ход для всех ваших объектов. Файлы классов Java содержат все типы данных и информацию о сигнатурах методов, которую содержат исходные файлы, поэтому компиляция бинарных файлов классов является такой же безопасной по отношению к типам (и по отношению к исключениям) как компиляция исходного кода Java.

Файлы JAR

Архивные файлы Java (JAR, Java Archive files) являются архивами файлов Java. Это стандартное кроссплатформенное средство упаковки всех частей вашего Java-приложения в компактную упаковку для распространения или установки. Вы можете положить в файл JAR все, что хотите: файлы классов Java, серийные объекты, файлы данных, изображения, аудио и др. Файл JAR может также иметь одну или несколько цифровых подписей, которые доказывают его целостность и подлинность. Подпись может прикрепляться к файлам в целом или индивидуальным предметам в файле.

Система выполнения Java может подгружать файлы прямо из архива в вашей переменной `CLASSPATH`, как описано ранее. Неклассовые файлы (данные, изображения и т. д.), содержащиеся в вашем файле JAR, могут также быть изъяты из пути к классу вашим приложением при использовании метода `getResource()` (описанного в главе 12). При использовании этого средства вашему коду нет необходимости знать, находится какой-либо ресурс в простом файле или входит в архив JAR. Является этот класс или данные элементом файла JAR или индивидуальным классом в пути к классу, вы всегда можете сослаться на него стандартным образом и позволить загрузчику классов Java определить местоположение.

Сжатие файлов

Элементы, хранящиеся в файлах JAR, сжимаются стандартной компрессией файлов ZIP. Сжатие делает загрузку классов через сеть намного быстрее. Быстрый анализ стандартного дистрибутива Java показывает, что типичный файл класса сокращается при сжатии на 40 процентов.

Такие текстовые файлы, как HTML или ASCII, содержащие английские слова, часто сжимаются до одной десятой от своего начального размера или меньше. (С другой стороны, файлы изображения обычно не становятся меньше при сжатии, поскольку распространенные форматы изображений сами являются форматами сжатия.)

Java также имеет формат архива, называющийся *Pack200,* который оптимизирован специально под байт-код класса Java и может достигнуть в четыре раза большей компрессии классов Java, чем просто ZIP. Мы поговорим о Pack200 позже в этой главе.

Утилита `jar`

Утилита `jar`, поставляемая с JDK, является простым инструментом для создания и чтения файлов JAR. Его интерфейс пользователя не очень-то дружелюбный. Он имитирует команду `tar` (хранилище магнитных лент) Unix. Если вы знакомы с `tar`, вы узнаете следующие магические формулы:

```
jar -cvf путь к jar-файлу [ путь ] [ ... ]
```

Создать *jar-файл*, содержащий *путь(и)*.

```
jar -tvf jar-файл [ путь ] [ ... ]
```

Вывести содержимое *jar-файла*, опционально показывая только *путь(и)*.

```
jar -xvf jar-файл [ путь ] [ ... ]
```

Извлечь содержимое *jar-файла*, опционально извлекая только *путь(и)*.

В этих командах буквы-маркеры *c*, *t* и *x* сообщают `jar`, создавать ли архив, перечислять ли содержимое архива или извлекать файлы из архива. *f* означает, что следующим аргументом является имя файла JAR, с которым нужно работать. Опциональная метка *v* говорит `jar` , что информацию о файлах нужно показывать в подробном виде. В режиме подробной информации вы получаете информацию о размерах файла, времени изменения и коэффициенте сжатия.

Последовательные элементы в командной строке (например, все, что находится за буквами, сообщающими `jar`, что делать, и файл, над

которым `jar` нужно производить действия) воспринимаются как имена элементов архива. (Если вы не выводите никакие файлы, `jar` извлекает все из архива.)

Представим, что мы только что закончили создание новой игры *spaceblaster*. Все файлы, связанные с игрой, находятся в трех папках. Сами классы Java находятся в папке *spaceblaster/game*, *spaceblaster/images* содержит изображения из игры и *spaceblaster/*

Docs содержит связанные данные игры. Мы можем упаковать все это в архив этой командой:

```
% jar -cvf spaceblaster.jar spaceblaster
```

Поскольку мы запросили подробный вывод, строка `jar` говорит, что она делает:

```
adding:spaceblaster/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/game/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/game/Game.class (in=8035) (out=3936) (deflated 51%)
adding:spaceblaster/game/Planetoid.class (in=6254) (out=3288) (deflated 47%)
adding:spaceblaster/game/SpaceShip.class (in=2295) (out=1280) (deflated 44%)
adding:spaceblaster/images/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/images/spaceship.gif (in=6174) (out=5936) (deflated 3%)
adding:spaceblaster/images/planetoid.gif (in=23444) (out=23454) (deflated 0%)
adding:spaceblaster/docs/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/docs/help1.html (in=3592) (out=1545) (deflated 56%)
adding:spaceblaster/docs/help2.html (in=3148) (out=1535) (deflated 51%)
```

`Jar` создает файл *spaceblaster*.jar и папку *spaceblaster*, добавляя папки и файлы, находящиеся в *spaceblaster* в архив. В режиме подробной информации `jar` сообщает об экономии, полученной от сжатия файлов в архив.

Мы можем распаковать архив этой командой:

```
% jar -xvf spaceblaster.jar
```

Аналогично, мы можем извлечь отдельный файл или директорию:

```
% jar -xvf spaceblaster.jar имяфайла
```

Но, конечно, обычно вам не нужно распаковывать файл JAR, чтобы использовать его содержимое; инструменты Java знают, как извлечь

файлы из архивов автоматически. Мы можем перечислить содержимое нашего архива JAR при помощи команды:

```
% jar -tvf spaceblaster.jar
```

Вот результат; он перечисляет все файлы, их размеры и их время создания:

```
0 Thu May 15 12:18:54 PDT 2003 META-INF/
1074 Thu May 15 12:18:54 PDT 2003 META-INF/MANIFEST.MF
0 Thu May 15 12:09:24 PDT 2003 spaceblaster/
0 Thu May 15 11:59:32 PDT 2003 spaceblaster/game/
8035 Thu May 15 12:14:08 PDT 2003 spaceblaster/game/Game.class
6254 Thu May 15 12:15:18 PDT 2003 spaceblaster/game/Planetoid.class
2295 Thu May 15 12:15:26 PDT 2003 spaceblaster/game/SpaceShip.class
0 Thu May 15 12:17:00 PDT 2003 spaceblaster/images/
6174 Thu May 15 12:16:54 PDT 2003 spaceblaster/images/spaceship.gif
23444 Thu May 15 12:16:58 PDT 2003 spaceblaster/images/planetoid.gif
0 Thu May 15 12:10:02 PDT 2003 spaceblaster/docs/
3592 Thu May 15 12:10:16 PDT 2003 spaceblaster/docs/help1.html
3148 Thu May 15 12:10:02 PDT 2003 spaceblaster/docs/help2.html
```

Манифесты JAR

Обратите внимание, что команда `jar` автоматически добавляет папку, называемую *META-INF,* к нашему архиву. Папка *META-INF* хранит файлы, описывающие содержание файла JAR. Она всегда содержит хотя бы один файл: *MANIFEST.MF.* Файл *MANIFEST.MF* может содержать «накладную», перечисляющую файлы в архиве, вместе с набором атрибутов для каждой записи, определяемым пользователем.

Манифест является текстовым файлом, содержащим набор строк в формате *ключевое слово: значение*. По умолчанию манифест пустой и содержит только информацию о версии файла JAR:

```
Manifest-Version: 1.0
Created-By: 1.7.0_07 (Oracle Corporation)
```

Также возможно подписать файлы JAR цифровой подписью. Когда вы это делаете, информационная сводка (контрольный итог) добавляется к манифесту для каждого элемента массива (как показано далее) и папка *META-INF* содержит файлы цифровой подписи для элементов архива.

```
Name: com/oreilly/Test.class
SHA1-Digest: dF2GZt8G11dXY2p4olzzIc5RjP3=
...
```

При создании архива вы можете дополнить описание манифеста собственным файлом. Это одно из возможных мест для хранения других простых типов неотъемлемой информации о файлах в архиве, такой как версия или сведения об авторстве.

Например, мы можем создать файл со следующими стоками в формате *ключевое слово: значение:*

```
Name: spaceblaster/images/planetoid.gif
RevisionNumber: 42.7
Artist-Temperament: moody
```

Чтобы добавить эту информацию в манифест вашего архива, поместите ее в файл, называемый *myManifest.mf*, и напишите следующую команду `jar`:

```
% jar -cvmf myManifest.mf spaceblaster.jar spaceblaster
```

Мы включили дополнительную опцию, `m`, которая указывает, что `jar` должен считывать дополнительную декларативную информацию из файла, указанного в командной строке. Как `jar` знает, где какой файл? Поскольку `m` находится перед `f`, он ожидает найти декларативную информацию перед именем файла JAR, который он создаст. Если вы думаете, что это странно, вы правы; стоит только поставить имена в неправильном порядке, `jar` все сделает неправильно. Приложение может получить содержимое манифеста, хранящегося в файле JAR, используя класс `java.util.jar.Manifest`.

Мы увидим больше примеров по добавлению информации в манифест JAR в главе 22. Интерфейс JavaBeans API использует информацию манифеста, чтобы определить, какие классы являются объектами, использующими атрибут `Java-Bean`. Эта информация используется IDE, которая работает с JavaBeans.

Создание выполняемого файла JAR

Кроме атрибутов вы можете поместить несколько специальных значений в файл манифеста. Один из них, `Main-Class`, позволяет вам определить исходный метод `main()` для приложения, содержащегося в JAR:

```
Main-Class: com.oreilly.Game
```

Если вы добавите это в ваш файл манифеста JAR (используя опцию `m`, описанную ранее), вы можете запустить приложение непосредственно из JAR:

```
% java -jar spaceblaster.jar
```

Что более важно, в операционной системе OS X, Windows и других средах с GUI, вы можете просто дважды щелкнуть мышью на файле JAR, чтобы запустить приложение. Интерпретатор ищет значение `Main-Class` в манифесте, затем загружает назначенный класс в качестве пускового класса для приложения.

Утилита pack200

Pack200 является форматом архива, который оптимизирован для хранения скомпилированных файлов классов Java. Pack200 не является новой формой сжатия, а скорее суперэффективным шаблоном для информации класса, устраняющим многие типы затрат и избытков в связанных классах. Это эффективно объемный формат файла класса, который разбирает многие классы и снова рационально собирает их части в один каталог. Затем это позволяет таким стандартным форматам сжатия, как ZIP, работать с максимальной эффективностью над архивом, достигая сжатия как минимум в четыре раза большего. Система выполнения Java не понимает формат Pack200, поэтому вы не можете поместить архивы этого типа в путь к классу. Наоборот, в основном средний формат очень выгоден для передачи JAR приложения по Интернету для апплетов или других типов интернет-приложений. Вы можете конвертировать JAR в формат Pack200 и наоборот при помощи команд `pack200` и `unpack200`, поставляемых с JDK.

Например, для конвертации *foo.jar* в *foo.pack.gz* нужно использовать команду:

```
% pack200 foo.pack.gz foo.jar
```

Чтобы конвертировать *foo.pack.gz* в *foo.jar:*

```
% unpack200 foo.pack.gz foo.jar
```

Отметьте, что процесс Pack200 полностью разрушает и реконструирует ваши классы на уровне класса, поэтому получаемый в результате файл *foo.jar* не будет байт-в-байт с исходным.

Файлы политики

Одна из действительно новых вещей в Java — это то, что безопасность встроена в язык. Как описано в главе 1, виртуальная машина Java может проверять файлы классов, а диспетчер безопасности Java может устанавливать пределы на действия классов. В ранних версиях Java было необходимо реализовывать политику безопасности *программным путем*, создавая файл диспетчера безопасности Java и используя его в вашем приложении. Позже была добавлена *декларированная* безопасность. Эта система позволяет вам писать *файлы политики* — текстовые описания полномочий, что намного проще и не требует изменения кода. Эти файлы политики говорят диспетчеру безопасности, что и кому можно позволять и запрещать.

В ранних версиях Java было много суеты вокруг безопасности апплетов. Апплеты, загруженные с сомнительных источников, могли запускаться с ограничениями по безопасности, которые запрещали им сомнительные вещи, такие как чтение с диска, запись на диск или контакт с произвольными компьютерами в сети. С файлами политики безопасности легко применять безопасность в виде апплета к любому приложению, не меняя его. Более того, легко совершенствовать гарантируемый вами доступ. Например, вы можете разрешить приложению доступ только к отдельной папке на диске или можете разрешить сетевой доступ к определенным адресам.

Понимание безопасности и политики безопасности может быть важным, поэтому мы рассмотрим эту тему. Однако на практике вы, скорее всего, не будете использовать эту возможность сами, если не пишете фреймворк для запуска приложения с многих неизвестных источников или не нуждаетесь в ограничении приложения по каким-то другим причинам.

Стандартный диспетчер безопасности

По умолчанию, когда вы локально запускаете приложение Java, диспетчер безопасности не установлен. Вы можете включить безопасность, используя опцию интерпретатора java для установки диспетчера безопасности по умолчанию. Политика безопасности по умолчанию осуществляет множество тех же правил, что и для апплетов. Чтобы посмотреть, как это работает, давайте напишем маленькую программу, которая делает что-то сомнительное: она производит сетевое соединение

с каким-то из компьютеров в сети. (Мы рассмотрим специфику сетевого программирования в главах 13 и 14.)

```
import java.net.*;
public class EvilEmpire {
public static void main(String[] args) throws Exception{
try {
Socket s = new Socket("207.46.131.13", 80);
System.out.println("Подключено!");
}
catch (SecurityException e) {
System.out.println("SecurityException: не удалось подключиться.");
}
}
}
```

Если вы запустите эту программу с помощью интерпретатора Java, она осуществит сетевое соединение:

```
C:\> java EvilEmpire
Подключено!
```

Но поскольку эта программа «плохая», давайте установим диспетчер безопасности по умолчанию таким образом:

```
C:\> java -Djava.security.manager EvilEmpire
SecurityException: не удалось подключиться.
```

Так лучше, но предположим, что у этого приложения действительно есть уважительная причина для совершения сетевого соединения. Мы бы хотели оставить диспетчер безопасности по умолчанию просто для спокойствия, но в то же время выдать этому приложению разрешение на совершение сетевого соединения.

Утилита `policytool`

Чтобы разрешить нашему примеру `EvilEmpire` совершать сетевое соединение, нам необходимо создать *файл политики*, который содержит необходимое разрешение. Включенная в JDK удобная утилита, называемая `policytool`, помогает создавать файлы политики. Запустите ее при помощи такой командной строки:

```
C:\> policytool
```

При запуске `policytool` вы можете получить сообщение об ошибке, гласящее, что файл политики по умолчанию не был найден. Не беспокойтесь об этом, просто нажмите ОК, чтобы сообщение исчезло.

Теперь мы добавляем сетевое разрешение для приложения `Evil Empire`. Приложение идентифицируется его происхождением, называемым также *кодовой базой*, описанной URL. В этом случае это `file:` адрес URL, который указывает на местоположение приложения `EvilEmpire` на вашем диске. Если вы запустили `policytool`, вы увидите его главное окно, показанное на рис. 3.2. Нажмите кнопку **Add Policy Entry** (Добавить запись о политике). Появится еще одно окно, как показано на рис. 3.3 (но с пустыми полями).

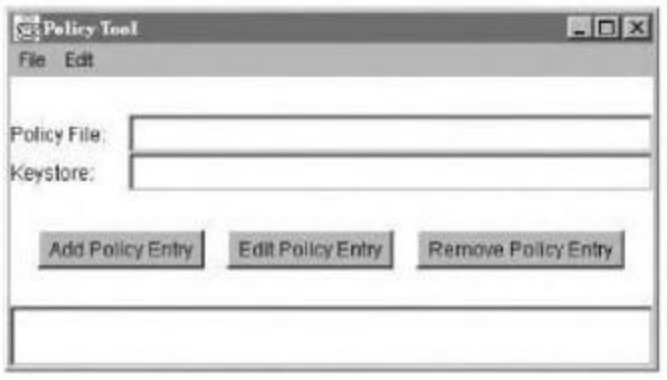

Рис. 3.2. Окно **Policy Tool**

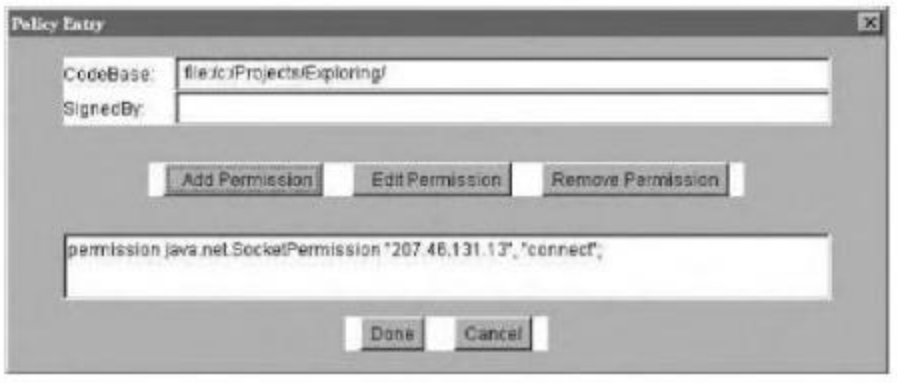

Рис. 3.3. Добавление записи о политике

Во-первых, введите URL-адрес в папки, содержащей EvilEmpire, в поле **Codebase**. Затем нажмите кнопку **Add Permission** (Добавить разрешение). Появится еще одно окно, как показано на рис. 3.4.

Выберите элемент **`SocketPermission`** из первого раскрывающегося списка. Затем во второе текстовое поле справа введите сетевой адрес, к которому подключится `EvilEmpire`. В завершение выберите элемент **`Connect`** из третьего раскрывающегося списка. Нажмите кнопку **ОК**; вы увидите новое разрешение в окне политики, как показано на рис. 3.3.

Нажмите кнопку **Done** (Готово), чтобы завершить создание политики. Затем выберите пункт **Save As** (Сохранить как) в меню **File** (Файл)

и сохраните политику с именем типа *EvilEmpire.policy*. Завешите работу утилиты `policytool`.



Рис. 3.4. Добавление разрешения

Файл политики, который мы сейчас создали, не сложен. Взгляните на него в текстовом редакторе, который показывает простой синтаксис созданной нами политики:

```
grant codeBase "file:/c:/Projects/Exploring/" {
permission java.net.SocketPermission "207.46.131.13", "connect";
};
```

Вы можете совсем отказаться от утилиты `policytool` и просто создать файлы политики при помощи текстового редактора, если вам так удобнее.

Использование файла политики со стандартным диспетчером безопасности

Теперь, раз уж мы взяли на себя труд создать файл политики, давайте используем его. Вы можете приказать диспетчеру безопасности по умолчанию использовать файл политики при помощи опции в командной строке интерпретатора `java`:

```
C:\> java -Djava.security.manager -Djava.security.policy=EvilEmpire.policy
EvilEmpire
Подключено!
```

`EvilEmpire` теперь может создать соединение с сокетом, потому что мы ясно дали ему разрешение при помощи файла политики. Однако диспетчер безопасности по умолчанию до сих пор защищает нас при помощи других способов. `EvilEmpire` не может записывать файлы на диск и считывать их, кроме файлов из той папки, из которой он сам, а также не может совершать соединений с другими сетевыми адресами, кроме указанного нами. Воспользуйтесь моментом и насладитесь этим теплым приятным чувством.

Глава 4
ЯЗЫК JAVA

С этой главы начинается наше знакомство с синтаксисом языка Java. Поскольку читатели приходят к этой книге с разными уровнями опыта в программировании, тяжело установить оптимальный уровень для всей аудитории. Мы постарались достигнуть равновесия между подробным описанием синтаксиса языка для начинающих и достаточным количеством справочной информации, чтобы опытный читатель мог быстро оценить разницу между Java и другими языками.

Поскольку синтаксис языка Java является производным от языка C, мы проводим некоторые сравнения со свойствами этого языка, но знание C не обязательно. Мы уделим больше внимания тем аспектам Java, которые отличают его от других языков, и меньше — идеям базового программирования. Например, мы подробно рассмотрим массивы в Java, поскольку они значительно отличаются от массивов в других языках. С другой стороны, мы не будем тратить много времени на объяснение базовых конструкций языка, таких как циклы и управляющие конструкции.

Главы с 5 по 7 будут строиться на основе этой главы, рассказывая об объектно-ориентированной стороне Java, и завершат рассмотрение базового языка. В главе 8 обсуждаются параметры настройки, свойство, которое улучшает работу типов в языке Java, позволяя вам писать определенные типы классов более гибко и безопасно. После этого мы погрузимся в API Java и посмотрим, что мы можем делать с языком. Остальная часть этой книги наполнена лаконичными примерами, которые делают полезные вещи, и если у вас остались какие-то вопросы после вступительных глав, мы надеемся, что вы получите ответы на них, когда ознакомитесь с кодом.

Кодировка текста

Язык Java активно применяется во Всемирной паутине. Поскольку пользователи Интернета говорят и пишут на многих человеческих

языках, Java также должна уметь управляться с большим количеством языков. Одним из путей, посредством которых Java поддерживает интернационализацию, является кодировка Unicode. Стандарт Unicode распространен во всем мире и поддерживает сценарии выполнения многих языков*. Символы и строковые данные последней версии Java основываются на стандарте Unicode 6.0, который использует хотя бы два байта для представления каждого символа изнутри.

Исходный код Java может быть написан при помощи Unicode и храниться в любом количестве кодировок символов, варьируясь от полностью бинарной формы до значений символов Unicode в кодировке ASCII. Это делает Java дружелюбным языком для программистов, не говорящих по-английски, которые могут использовать свой родной язык для имен классов, методов и переменных, как и для текста в приложении.

Тип `char` и класс `String` исходно поддерживают значения Unicode. Изнутри текст хранится в формате многобайтных символов с использованием кодировки UTF-16; однако язык Java и API делают их невидимыми для вас, и, как правило, вам не придется об этом думать. Unicode также очень дружелюбен к ASCII (ASCII является самой известной кодировкой для английского языка). Первые 256 символов определены как идентичные первым 256 символам в наборе символов 8859-1 (Латынь-1), поэтому Unicode является эффективно обратно совместимым с самыми распространенными английскими наборами символов. Более того, одна из самых распространенных кодировок файла для Unicode, называемая UTF-8, сохраняет значения ASCII в их однобайтной форме. Эта кодировка используется по умолчанию в компилированных файлах классов Java, поэтому хранение остается компактным для английского текста.

Большинство платформ не могут отобразить все определенные сейчас символы Unicode. Как результат, программы Java могут быть написаны с особыми последовательностями переключения кода Unicode. Символ Unicode может быть представлен следующей последовательностью переключения:

```
\uxxxx
```

---

* Дополнительную информацию о Unicode ищите по адресу **www.unicode.org**. По иронии судьбы, одним из скриптов, который указан как «устаревший и архаичный» и в данный момент не поддерживается стандартом Unicode, является яванский — исторический язык людей с острова Ява (Java).

«xxxx» является последовательностью шестнадцатеричных цифр один к четырем. Последовательность переключения указывает на символ Unicode в кодировке ASCII. Это также является формой, используемой Java для вывода (печати) символов Unicode в среде, которая другим образом их не поддерживает. Java также поставляется с классами для чтения и написания потоков символов Unicode в специфической кодировке, включая UTF-8.

Комментарии

Java поддерживает *комментарии блока* в стиле языка C, ограниченные символами /* и */, и *строчные комментарии* языка C++, обозначенные символами //:

```
/* Это
Многострочный
комментарий */
// Это однострочный комментарий
// и это // тоже
```

Комментарии блока начинаются и заканчиваются последовательностью символов и могут охватить большой диапазон текста. Однако они не могут быть «вложенными», что означает, что один блочный комментарий не может находиться в другом блочном комментарии, не приводя в замешательство компилятор.

Однострочные комментарии имеют только начальные последовательности и ограничиваются концом строки; дополнительные символы // внутри одной строки не имеют эффекта. Строчные комментарии используются для коротких комментариев внутри методов; они не конфликтуют с блочными комментариями, поэтому вы можете снабжать комментарием большие куски кода, в которые они вложены.

Комментарии Javadoc

Блочный комментарий, начинающийся с символов /**, означает специальный *комментарий документа*. Комментарий документа разработан для извлечения автоматическими генераторами документа, такими как JDK-программа `javadoc`. Комментарий документа заканчиваются символами */, как и в обычном блочном комментарии. Внутри комментария документа строки, начинающиеся с `@`, интерпретируются

как специальные инструкции для генерации документации, давая ему информацию об источнике кода. Обычно каждая строка комментария документа начинается с *, как показано в следующим примере, но это опционально. Любой предшествующий пробел и * в конце каждой строки игнорируется:

```
/**
* Я думаю, этот класс — вероятно, самая поразительная вещь,
* которую вы когда-либо видели. Я расскажу вам о своем
* собственном видении и мотивации при его создании.
* <p>
* Это все началось, когда я был маленьким ребенком, который
* рос на улицах Айдахо. Все были без ума от картофеля,
* и жизнь была хороша…
*
* @see PotatoPeeler
* @see PotatoMasher
* @author John 'Spuds' Smith
* @version 1.00, 19 Dec 2006
*/
class Potato {
```

Javadoc создает документацию HTML для классов чтением исходного кода, вытаскивая встроенные комментарии и теги @.

В этом примере теги являются причиной того, что автор и информация о версии представлены в документации класса. Теги @see производят гиперссылки на связанную документацию класса.

Компилятор также смотрит на комментарии документа; в частности он заинтересован в теге @deprecated, который означает, что метод был объявлен устаревшим и его нужно избегать в новых программах. Тот факт, что метод исключен, отмечается в компилированном файле класса, поэтому предупреждение может генерироваться каждый раз, когда вы используете устаревшее свойство в вашем коде (даже если источник недоступен).

Комментарии документа могут появляться над классом, методом и определениями переменных, но некоторые теги могут быть неприменимы ко всему этому. Например, тег @exception может применяться только к методам. Табл. 4.1 резюмирует теги, используемые в комментариях документа.

Табл. 4.1. Теги, используемые в комментариях документа

Тег	Описание	Применяется к
`@see`	Связанное имя класса	Класс, метод или переменная
`@author`	Имя автора	Класс
`@version`	Вариант сроки	Класс
`@param`	Имя и описание параметра	Метод
`@return`	Описание значения возврата	Метод
`@exception`	Имя и описание исключения	Метод
`@deprecated`	Объявляет элемент устарелым	Класс метод или переменная
`@since`	Заявляет версию API, когда элемент добавлен	Переменная

Javadoc как метаданные

Теги Javadoc в комментариях документа представляют *метаданные* об источнике кода; то есть они добавляют пояснения о структуре или содержании кода, который, строго говоря, не является частью приложения. Некоторые дополнительные инструменты расширяют понятие тегов в стиле Javadoc для включения других типов метаданных о программах Java, которые содержатся в компилированном коде и могут с готовностью быть использованы приложением, чтобы воздействовать на его компиляцию или поведение во время выполнения. Возможность аннотации Java обеспечивает более формальный и расширяемый способ добавления метаданных к классам, метода и переменным Java. Мы поговорим об *аннотациях* в главе 7. Однако мы должны упомянуть, что существует аннотация `@deprecated`, которая имеет то же значение, что и одноименный тег Javadoc, и вы, возможно, предпочтете использовать ее.

Типы

Система типа в языке программирования описывает, как его элементы данных (переменные и константы) связаны с хранением в памяти и как связаны друг с другом. В статическом языке, таком как C и C++, тип элемента данных является простым, неизменным атрибутом, который часто соответствует непосредственно некоему лежащему в основе явлению аппаратного обеспечения, такому как реестр или значение указателя. В более динамическом языке, таком как Smalltalk или Lisp,

переменные могут быть назначенными произвольными элементами и эффективно менять свой тип во время своего время действия. Значительное количество расходов идет на проверку того, что происходит в этих языках во время выполнения. Языки сценариев, подобные Perl, достигают простоты использования, предоставляя радикально упрощенные системы типов, в которых только определенные элементы данных могут храниться в переменных, а значения объединены в такое общее представление, как строка.

Язык Java объединяет многие лучшие свойства статических и динамических языков. Как и в статическом языке, каждая переменная и элемент программирования в Java имеют тип, который известен во время компиляции, поэтому системе выполнения обычно не приходится проверять назначения между типами во время выполнения кода. В отличие от традиционных языков C и C++ Java также управляет информацией об объектах во время выполнения и использует это, чтобы позволить действительно динамическое поведение. Код Java может загружать новые типы во время выполнения и использовать их в полностью объектно-ориентированных способах, позволяя преобразование и полный полиморфизм (расширение типов). Код Java может также размышлять над собственными типами во время выполнения или исследовать их, позволяя улучшенные типы поведения приложения, такие как интерпретаторы, которые могут взаимодействовать с компилированными программами динамически.

Типы данных в Java делятся на две категории. *Простые типы* представляют простые значения, которые имеют встроенную функциональность в языке; они являются фиксированными элементами, такими как буквенные константы и числа. *Ссылочные типы* (или классовые типы) включают объекты и массивы; они называются ссылочными типами, поскольку они «ссылаются на» большие типы данных, которые передаются «по ссылке», как мы объясним вскоре. *Обобщенные* типы в действительности являются композицией (объединением) типов класса, и соответственно, являются также ссылочными типами.

Простые типы

Числа, символы и булевы значения являются базовыми элементами в Java. В отличие от некоторых других (возможно, более чистых) объектно-ориентированных языков, в Java они не являются объектами. Для таких ситуаций, когда желательно обращаться с простыми значе-

ниями как с объектами, Java предоставляет классы «обертки». Главным преимуществом обращения с простыми значениями как с объектами является то, что компилятор Java и среда выполнения могут с большей готовностью оптимизировать их реализацию. Простые значения и вычисления могут осуществляться программно-аппаратными средствами, как это всегда было в машинно-ориентированных языках. Далее мы увидим, как Java может автоматически конвертировать простые значения и их объектные обертки так, чтобы частично замаскировать разницу между ними двумя. Мы объясним, что это значит, более подробно в следующей главе, когда будем обсуждать *упаковку* и *распаковку* простых значений.

Важным портативным свойством Java является то, что простые значения точно определены. Например, вам никогда не придется беспокоиться о размере типа `int` на определенной платформе; он всегда равен 32-битному относительному числу в дополнительном двоичном коде. Табл. 4.2 обобщает простые типы Java.

Табл. 4.2. Простые типы Java

Тип	Определение
`boolean`	`true` или `false`
`char`	16-битный символ Unicode
`byte`	8-битный, относительный, представленный дополнительным двоичным целым значением
`short`	16-битный, относительный, представленный дополнительным двоичным целым значением
`int`	32-битный, относительный, представленный дополнительным двоичным целым значением
`long`	64-битный, относительный, представленный дополнительным двоичным целым значением
`float`	32-битное IEEE 754, значение с плавающей запятой
`double`	64-битное, IEEE 754



Те из вас, у кого есть опыт программирования на языке Си, могут заметить, что простые типы выглядят как упрощенные скалярные типы языка Си на 32-битной машине, и вы будете абсолютно правы. Это то, как они должны выглядеть. Употребление 16-битных символов обусловлено стандартами Unicode, и ситуативные указатели были удалены по другим причинам. Но вообще синтаксис и семантика простых типов Java произошли от Си.

Точность операций с плавающей запятой

Операции с плавающей запятой в Java следуют международной спецификации IEEE 754 — это означает, что результат вычислений плавающей запятой обычно один и тот же на разных платформах Java. Однако Java предусматривает расширенную точность на платформах, которые ее поддерживают. Это может внести скрытую разницу результатов прецизионных операций в очень маленьких значениях. Большинство приложений никогда не заметят этого, но если вы хотите гарантировать, что ваши вычисления будут иметь абсолютно одинаковые результаты на разных платформах, вы можете использовать специальное ключевое слово `strictfp` как модификатор класса на классе, содержащем манипуляцию с плавающей точкой (мы рассмотрим классы в следующей главе). Тогда компилятор запретит эти оптимизации под определенные платформы.

Объявление и инициализация переменной

Переменные объявляются внутри методов и классов с названием типа, за которым следует одно или больше имен переменных, разделенных запятой. Например:

```
int foo;
double d1, d2;
boolean isFun;
```

Переменные можно при желании инициализировать выражением подходящего типа, когда они объявляются:

```
int foo = 42;
double d1 = 3.14, d2 = 2 * 3.14;
boolean isFun = true;
```

Переменным, которые объявляются как члены класса, устанавливаются значения по умолчанию, если они не инициализированы (см. главу 5). В этом случае числовые типы приравниваются к подходящей разновидности ноля, символам устанавливается нолевой символ (`\0`), а булевы переменные имеют значение `false`. Локальные переменные, которые объявляются внутри метода и существуют только во время вызова метода, с другой стороны, должны быть ясно инициализированы перед употреблением. Как мы увидим, компилятор обеспечивает соблюдение этого правила, поэтому вы не рискуете забыть его.

Целочисленные литералы

Целочисленные литералы могут быть указаны в восьмеричной (с основанием 8), десятичной (с основанием 10), шестнадцатеричной (с основанием 16) системах. Десятичное целое указывается последовательностью цифр, начинающейся одним из символов от 1 до 9:

```
int i = 1230;
```

Восьмеричные числа отличаются от десятичных тем, что у них в начале 0:

```
int i = 01230; // i = 664 в десятичной системе
```

Шестнадцатеричное число обозначается стоящим впереди значением 0x или 0X (ноль «x»), за которым следует комбинация цифр и символов a — f или A — F; что представляет десятичные значения 10 — 15:

```
int i = 0xFFFF; // i = 65535 в десятичной системе
```

Целочисленные литералы имеют тип int, если за ними не стоит суффикс L, обозначающий, что они должны стать длинным значением:

```
long l = 13L;
long l = 13; // эквивалент: тип int превращен 13
```

(Буква l в нижнем регистре также применима, но нужно этого избегать, поскольку она часто выглядит как цифра 1.)

Когда числовой тип используется в назначении или выражении, включающем «больший» тип с большим диапазоном, он может быть *продвинут* до большего типа. Во второй строке предыдущего примера число 13 имеет дефолтный тип int, но оно продвигается до типа long, поскольку он связан с переменной типа long. Некоторые другие числовые операции и операции сравнения также вызывают этот тип арифметического продвижения, как делают математические выражения, включающие более одного типа. Например, при умножении байтового значения на значение int компилятор вначале продвигает тип byte до int:

```
byte b = 42;
int i = 43;
int result = b * i; // b продвигается до int перед умножением
```

Числовой тип никогда не может стать другим типом и быть назначен типу с меньшим диапазоном без точного приведения, однако:

```
int i = 13;
byte b = i; // ошибка времени компиляции, необходимо точное приведение
типа
byte b = (byte) i; // OK
```

Преобразование из плавающего типа в целый всегда нуждается в точном приведении из-за потенциальной потери точности. Наконец мы должны отметить, что, если вы используете Java версии 7 или старше, вы можете добавить немного форматирования к вашим числовым литералам, используя символ нижнего подчеркивания «_» между цифрами. Так что если у вас есть очень большие строки с цифрами, вы можете прерывать их как в следующих примерах:

```
int RICHARD_NIXONS_SSN = 567_68_0515;
int for_no_reason = 1___2___3;
int JAVA_ID = 0xCAFE_BABE;
```

Нижнее подчеркивание может возникнуть только между цифрами, не в начале или конце числа или рядом со знаком L, означающим длинное целое.

Литералы с плавающей запятой

Значения с плавающей запятой могут быть указаны в десятичной или экспоненциальной записи. Литералы с плавающей запятой относятся к типу `double`, если за ними не следует суффикс `f` или `F`, обозначающий, что они должны быть созданы как значения типа `float`. И также, как с целочисленными литералами, вы можете использовать символ нижнего подчеркивания «_» в Java 7 для создания чисел с плавающей запятой — но только между цифрами, не в начале, конце или рядом с точкой в десятичной дроби или знаком F в числе.

```
double d = 8.31;
double e = 3.00e+8;
float f = 8.31F;
float g = 3.00e+8F;
float pi = 3.14_159_265_358;
```

Двоичные литералы

Новым свойством Java 7 является введение значений двоичных литералов. Это позволяет вам выполнять контрольное считывание бинар-

ных значений, непосредственно поставив префикс `0b` или `0B` (ноль B) перед числом.

```
byte one = (byte)0b00000001;
byte two = (byte)0b00000010;
byte four = (byte)0b00000100;
byte sixteen = (byte)0b00001000;
int cafebabe = 0b11001010111111101011101010111110;
long lots_o_ones = (long)0b11111111111111111111111111111111111111111111111
1L;
```

Символьные литералы

Значение символьного литерала может быть указано как символ с одной кавычкой или как управляющая последовательность ASCII:

```
char a = 'a';
char newline = '\n';
char smiley = '\u263a';
```

Ссылочные типы

В таком объектно-ориентированном языке, как Java, вы можете создавать новые, комплексные типы данных из простых типов, создав класс. Каждый класс затем служит новым типом в языке. Например, если мы создадим новый класс в Java, называющийся `Foo`, мы также косвенно создадим новый тип, называющийся `Foo`. Тип элемента управляет тем, как он используется и где он может быть назначен. Что касается простых типов, элемент типа `Foo` может в общем быть назначен переменной типа `Foo` или передан как аргумент методу, который принимает значение `Foo`.

Тип не является простым атрибутом. Классы могут иметь отношения с другими классами и так же ведут себя типы, которые они представляют. Все классы в Java существуют в иерархии родитель-ребенок, где дочерний класс *подкласса* является специализированным типом своего родительского класса. Соответствующие типы имеют такие же отношения, где тип дочернего класса считается подтипом родительского класса. Поскольку дочерние классы наследуют всю функциональность их родительских классов, объект дочернего типа может использоваться вместо объекта родительского типа. Например, если вы создадите но-

вый класс Cat, который расширяет Animal, новый тип Cat считается подтипом Animal. Объекты типа Cat затем могут использоваться везде, где может использоваться объект типа Animal; говорится, что объект типа Cat может быть назначен переменной типа Animal. Это называется *полиморфизмом подтипа* и является одним из первичных свойств объектно-ориентированного программирования. Мы рассмотрим более подробно классы и объекты в главе 5.

Простые типы в Java используются и передаются «по значению». Другими словами, когда простое значение, как int, назначается переменной или передается аргументу или методу, оно просто копируется. *Ссылочные* типы (типы классов), с другой стороны, всегда достигаются «по ссылке». Ссылка является просто основой или именем объекта. Переменная ссылочного типа содержит «указатель» на объект этого типа (или подтипа, как обсуждалось ранее). Когда ссылка назначается переменной или передается методу, копируется только ссылка, но не объект, на который она указывает. Ссылка похожа на указатель в языке C или C++, если ее тип не установлен принудительно. Само значение ссылки не может быть открыто, создано или изменено. Переменная приобретает значение ссылки только при назначении определенному объекту.

Давайте запустим пример. Мы объявляем переменную типа Foo, называемую myFoo, и назначаем ее нужному объекту*:

```
Foo myFoo = new Foo();
Foo anotherFoo = myFoo;
```

myFoo является переменной ссылочного типа, которая содержит ссылку на только что созданный объект Foo. (Сейчас не волнуйтесь о подробностях создания объекта; мы обсудим это в главе 5.) Мы объявляем вторую переменную типа Foo, anotherFoo, и назначаем ее тому же объекту. Теперь у нас есть две идентичные ссылки: myFoo и anotherFoo, но один фактический экземпляр объекта Foo. Если мы будем менять что-то в состоянии объекта Foo, мы увидим тот же эффект, смотря на него с любой ссылки.

Ссылки объекта передаются методу таким же образом. В этом случае и myFoo, и anotherFoo будут служить эквивалентными аргументами:

```
myMethod( myFoo );
```

---

* Для сравнения код на C++ будет таким:

```
Foo& myFoo = *(new Foo());
Foo& anotherFoo = myFoo;
```

Здесь нужно сделать важное, но иногда сбивающее с толку разграничение, которое заключается в том, что сама ссылка является значением и что значение копируется, когда оно назначается переменной или передается во время вызова метода. Принимая во внимание наш предыдущий пример, аргумент, передаваемый методу (локальной переменной с точки зрения метода), в действительности является третьей ссылкой на объект `Foo`, в дополнение к `myFoo` и `anotherFoo`. Метод может менять состояние объекта `Foo` посредством этой ссылки (вызывая его методы или меняя его переменные), но он не может изменить понятие вызывающей программы о ссылке на `myFoo`, а именно: метод не может сделать, чтобы `myFoo` в вызывающей программе указывало на другой объект `Foo`; он может изменить только собственную ссылку. Это станет более очевидным, когда мы будем говорить о методах позже. В этом отношении Java отличается от C++. Если вам нужно изменить ссылку вызывающей программы на объект в Java, вам необходим дополнительный уровень использования косвенной адресации. Вызывающей программе придется обернуть ссылку в другой объект, так что оба смогут делить ссылку на него.

Ссылочные типы всегда указывают на объекты, а объекты всегда определяются классами. Однако два особых типа ссылок — массивы и интерфейсы — определяют тип объекта, на который они указывают, немного другим образом.

Массивы в Java занимают особое место в системе типов. Они являются особым типом объекта, создающимся автоматически, чтобы содержать коллекцию какого-то другого типа объекта, известного как *базовый тип*. Объявление ссылки массивного типа косвенно создает новый тип класса, разработанный как контейнер для базовых типов, что вы увидите в следующей главе.

Интерфейсы немного коварнее. Интерфейс определяет набор методов и дает ему соответствующий тип. К объекту, который реализует этот метод интерфейса, можно ссылаться при помощи этого типа интерфейса так, же, как и из его типа. Переменные и аргументы метода могут быть объявлены типами интерфейса, как и другие типы классов и любой объект, который реализует интерфейс, может быть назначен им. Это добавляет гибкости системе типов и позволяет Java выходить за границы иерархии классов и создавать объекты, которые могут эффективно иметь много типов. Мы рассмотрим интерфейсы в следующей главе.

Обобщенные типы, или *параметризированные типы*, как мы упоминали ранее, являются расширением синтаксиса классов Java, который

создает возможности для дополнительной абстракции в том, как классы работают с другими типами Java. Обобщенные типы делают возможной специализацию классов пользователем без изменения исходного кода класса. Мы рассмотрим обобщенные типы в подробностях в главе 8.

Слово о строках

Строки в Java представляют собой объекты таким образом, они являются ссылочным типом. Объекты класса `String`, однако, получают особую помощь от компилятора Java, который делает их больше похожими на простые типы. Литеральные значения строк в исходном коде Java превращаются в объекты класса `String` при помощи компилятора. Они могут использоваться прямо, передаваться как аргументы методам или назначаться переменным типа `String`:

```
System.out.println( "Hello, World..." );
String s = "I am the walrus...";
String t = "John said: \"I am the walrus...\"";
```

Символ `+` в Java «перегружен» для выполнения сцепления строк, а также для обычного добавления чисел. Вместе с `+=` он является единственным перегруженным оператором в Java:

```
String quote = "Four score and " + "seven years ago,";
String more = quote + " our" + " fathers" + " brought...";
```

Java создает один объект типа `String` из связанных строк и предоставляет его как результат выражения. Мы подробно обсудим класс `String` и все, что связано с текстом, в главе 10.

Операторы и выражения

Операторы Java появляются внутри методов и классов; они описывают всю деятельность программы Java. Объявления и назначения переменных, такие, как в предыдущей главе, являются операторами, как и базовые структуры языка, такие, как условный оператор `if/then` и циклы.

```
int size = 5;
if ( size > 10 )
doSomething();
for( int x = 0; x < size; x++ ) { ... }
```

Выражения производят значения; выражение производит результат, который можно использовать как часть другого выражения или в операторе. Вызовы метода, размещение объектов и, конечно, математические выражения являются примерами выражений. Формально, поскольку назначения переменных могут использоваться как значения для дальнейших назначений или операций (в немного сомнительном стиле программирования), они могут считаться операторами и выражениями.

```
new Object();
Math.sin( 3.1415 );
42 * 64;
```

Одна из установок Java — все должно быть простым и целостным. Исходя из этого, когда нет других ограничений, вычисление и инициализация в Java всегда происходят в том же порядке, в котором они возникают в коде, — слева направо, сверху вниз. Мы увидим использование этого правила в вычислении выражений присваивания, вызовах методов, индексах массивов. В некоторых других языках порядок вычисления более сложный или даже зависимый от реализации. Java избавляется от этого источника опасности, точно и просто определяя то, как вычисляется код. Это, однако, не означает, что вы должны начать писать малопонятные и витиеватые операторы. Полагаться на порядок вычисления выражений сложными путями — плохая привычка для программиста, даже если это работает. Это производит код, который тяжело читать и еще тяжелее редактировать.

Операторы

Операторы и выражения в Java появляются в *блоке кода*. Синтаксически блок кода является серией операторов, заключенных в открывающую фигурную скобку (`{`) и закрывающую фигурную скобку (`}`). Операторы в блоке кода могут включать объявление переменных и большинство операторов и выражений, которые мы упомянули ранее:

```
{
int size = 5;
setName("Max");
...
}
```

Методы, имеющие вид функции C, в каком-то смысле являются просто блоками кода, которые принимают параметры и могут вызываться их именами — например, метод `setUpDog()`:

```
setUpDog( String name ) {
int size = 5;
setName( name );
...
}
```

Объявление переменных ограничено до содержащего их блока кода, то есть их невозможно увидеть вне ближайшего набора скобок:

```
{
int i = 5;
}
i = 6; // ошибка во время компиляции, нет такой переменной i
```

Таким образом, блоки кода могут использоваться для произвольной группировки других операторов и переменных. Однако наиболее часто применение блоков кода заключается в определении группы операторов для использования в условном или итеративном операторе.

Условный оператор `if/else`

Мы можем определить условие `if/else` следующим образом:

```
if ( условие )
оператор;
[ else
оператор; ]
```

(Весь предыдущий пример сам является оператором и может быть помещен в другое условие `if/else`.)

Условие `if` имеет общую функцию принятия двух разных форм: «однострочной» или блочной. Блочная форма имеет следующий вид:

```
if ( условие ) {
[ оператор; ]
[ оператор; ]
[ ... ]
} else {
[ оператор; ]
[ оператор; ]
[ ... ]
}
```

Условие является булевым выражением. Булево выражение — это значение `true` или `false` либо выражение, результатом вычисления которого является одно из этих значений. Например, `i == 0` является булевым выражением, которое проверяет, содержит ли 0 целое `i`.

Во второй форме операторы содержатся в кодах блока, и все заключающиеся в них операторы выполняются, если соответствующая ветка (`if` или `else`) выполняется. Все переменные, объявляемые внутри каждого блока, видны только операторам внутри блока. Подобно условию `if/else`, большинство остальных операторов Java имеет отношение к управлению потоком выполнения. По большому счету они ведут себя как их тезки в других языках.

Циклы do/while

Операторы цикла `do` и `while` имеют знакомую функциональность; их проверка на условие также является булевым выражением:

```
while ( условие )
оператор;
do
оператор;
while ( условие );
```

например:

```
while( queue.isEmpty() )
wait();
```

В отличие от циклов `while` или `for` (их мы увидим далее), которые сначала проверяют их условия, цикл `dowhile` всегда хотя бы раз выполняет свои операторы.

Цикл for

Самая общая форма цикла `for` также унаследована от языка C:

```
for ( инициализация; условие; инкрементор )
оператор;
```

Раздел инициализации переменной может объявлять или инициализировать переменные, которые ограничены рамками оператора `for`. Цикл `for` затем начинает возможную серию кругов, в которых сначала

проверяется условие и, если оно истинно, тело оператора (или блок) выполняется. Следуя за каждым выполнением тела, выражения с инкрементарами вычисляются, чтобы дать им возможность обновлять переменные перед началом следующего круга:

```
for ( int i = 0; i < 100; i++ ) {
System.out.println( i );
int j = i;
...
}
```

Этот цикл выполнится 100 раз, выводя значения от 0 до 99. Заметьте, что переменная `j` является локальной для блока (видимой только операторам внутри него) и не будет доступна для кода «после» цикла `for`. Если условие цикла `for` возвращает `false` при первой проверке, тело и часть с инкрементором никогда не выполнятся.

Вы можете использовать множество разделенных запятой выражений в инициализации и части с инкрементором цикла `for`. Например:

```
for (int i = 0, j = 10; i < j; i++, j-- ) {
...
}
```

Вы также можете инициализировать существующие переменные вне границ цикла `for` в блоке инициализатора. Вы могли бы это делать, если бы хотели использовать конечное значение переменной цикла где-то в другом месте:

```
int x;
for( x = 0; hasMoreValue(); x++ )
getNextValue();
System.out.println( x );
```

Улучшенный цикл for

Цикл Java, благосклонно названный «улучшенным циклом `for`», ведет себя как оператор `foreach` в некоторых других языках, выполняющий итерацию над серией значений в массиве или другим типом коллекции:

```
for ( varDeclaration : iterable )
оператор;
```

Улучшенный цикл `for` может использоваться для прохождения по массивам любого типа, а также объектам Java любого типа, которые реализуют интерфейс `java.lang.Iterable`. Это включает классы API Библиотек Java. Мы поговорим о массивах в этой и следующей главах; в главе 11 говорится о библиотеках Java. Ниже представлена пара примеров:

```
int [] arrayOfInts = new int [] { 1, 2, 3, 4 };
for( int i : arrayOfInts )
System.out.println( i );
List<String> list = new ArrayList<String>();
list.add("foo");
list.add("bar");
for( String s : list )
System.out.println( s );
```

Опять же, мы не обсуждали массивы или класс `List` и специальный синтаксис в этом примере. То, что мы показываем здесь, — это улучшенный цикл `for`, проводящий итерации над массивом целых чисел, а также списком строчных переменных. Во втором случае `List` реализует интерфейс `Iterable` и таким образом может быть целью цикла `for`.

Операторы switch

Наиболее распространенная форма оператора `switch` Java принимает аргумент целого типа (или аргумент числового типа, который автоматически может «продвинуться» до целого типа), строчного типа или типа `enum` (вскоре обсудим его) и выбирает между рядом альтернативных, постоянных веток `case`*:

```
switch ( выражение )
{
case constantExpression :
оператор;
[ case constantExpression :statement; ]
...
[ default :
оператор; ]
}
```

Оператор `case` для каждой ветви должен вычислять константное целое число или значение строки во время компиляции. Строки срав-

* Строки в операторы `switch` были добавлены в Java 7.

ниваются при использовании метода `equals()`, класса `String`, который мы обсудим подробнее в главе 10. Опционный `case default` может быть указан для отлова несовпадающих условий. При выполнении `switch` просто находит ветку, соответствующую ее условному выражению (или дефолтной ветке) и выполняет соответствующий оператор. Но это еще не конец истории. Вопреки ожиданиям, оператор `switch` затем продолжает выполнять ветки после совпавших веток, пока не достигнет конца `switch` или особого оператора, называющегося `break`. Ниже представлена пара примеров:

```
int value = 2;
switch( value ) {
case 1:
System.out.println( 1 );
case 2:
System.out.println( 2 );
case 3:
System.out.println( 3 );
}
// выводит 2, 3!
```

Более распространенным является использование `break` для завершения каждой ветки:

```
int retValue = checkStatus();
switch ( retVal )
{
case MyClass.GOOD :
// что-то хорошее
break;
case MyClass.BAD :
// что-то плохое
break;
default :
// ни то, ни другое
break;
}
```

В этом примере только одна ветка — `GOOD`, `BAD` или дефолтная — выполняется. Поведение «срыва» `switch` обоснованно, когда вы хотите обработать несколько возможных значений `case` одним оператором, не прибегая к куче операторов `if/else`:

```
int value = getSize();
switch( value ) {
case MINISCULE:
case TEENYWEENIE:
case SMALL:
System.out.println("Small" );
break;
case MEDIUM:
System.out.println("Medium" );
break;
case LARGE:
case EXTRALARGE:
System.out.println("Large" );
break;
}
```

Этот пример эффективно группирует шесть возможных значений в три блока.

Перечисление и операторы switch

Перечисления предназначены, чтобы заменить использование целочисленных констант для таких ситуаций, как та, что мы обсуждали только что, обеспечивая типовую безопасность. Перечисления используют объекты как их значения вместо целых чисел, но сохраняют понятие расстановки и сопоставимости. В главе 5 мы увидим, что перечисления объявляются также как классы и что значения могут «импортироваться» в код вашего приложения и использоваться как константы. Например:

```
enum Size { Small, Medium, Large }
```

Вы можете использовать перечисления в операторах выбора так же, как использовали целочисленные константы в предыдущих примерах с операторами выбора. В действительности намного безопаснее так делать, поскольку перечисления имеют действительные типы, а компилятор не позволяет вам ошибочно добавлять блоки, которые не подходят никаким значениям, или смешивать значения разных перечислений.

```
// использование
Size size = ...;
switch ( size ) {
case Small:
```

```
...
case Medium:
...
case Large:
...
}
```

В главе 5 предоставлено больше информации о перечислениях.

Операторы break/continue

Оператор `break` в Java и его коллега `continue` могут также использоваться, чтобы оборвать цикл или условный оператор, выскакивая из него. `Break` заставляет Java остановить текущий оператор блока и возобновить выполнение после этого. В следующем примере цикл `while` продолжается бесконечно, пока метод `condition()` не вернет значение `true`, запуская оператор `break`, который останавливает цикл и продолжает с места, обозначенного как «после `while`».

```
while( true ) {
if ( condition() )
break;
}
// после while
```

Оператор `continue` заставляет операторы `for` и `while` перейти к следующей итерации, возвращаясь к тому месту, где они проверяют их условие.

Следующий пример выводит цифры от 0 до 99, пропуская номер 33.

```
for( int i=0; i < 100; i++ ) {
if ( i == 33 )
continue;
System.out.println( i );
}
```

Операторы `break` и `continue` выглядят как аналогичные операторы в языке C, но формы Java имеют дополнительную способность принимать ярлык в качестве аргумента и выскакивать из многократных уровней до рамок помеченного пункта в коде. Их использование не очень распространено в повседневном программировании на Java, но может быть важно в особых случаях. Вот пример:

```
labelOne:
while ( условие ) {
...
labelTwo:
while ( условие ) {
...
// точка прерывания или продолжения
}
// после labelTwo
}
// после labelOne
```

Ограничивающие операторы, скажем, блоки кода, условные операторы и циклы, могут быть помечены такими идентификаторами, как `labelOne` и `labelTwo`.

В этом примере `break` или `continue` без аргумента в указанной позиции имеет тот же эффект, что и в предыдущих примерах. `Break` вызывает продолжение вычисления в месте, обозначенном «после `labelTwo`»; `continue` немедленно заставляет цикл `labelTwo` вернуться к проверке условия.

Оператор `break labelTwo` в указанном месте оказывает тот же эффект, что и обычный `break`, но `break labelOne` прерывает оба уровня и возобновит в месте, обозначенном «после `labelOne`». Подобным образом `continue labelTwo` служит как обычный `continue`, но `continue labelOne` возвращает к проверке цикла `labelOne`. Многоуровневые операторы `break` и `continue` устраняют основной недостаток согласования оператора `goto` в C/C++.

Есть несколько операторов Java, которые мы не будем рассматривать прямо сейчас. Операторы `try`, `catch` и `finally` используются в обработке исключений, что мы обсудим далее в этой главе. Оператор `synchronized` в Java используется для согласования доступа к операторам среди многочисленных потоков выполнения; смотрите главу 9, где рассказывается о синхронизации потоков.

Недостижимые операторы

Наконец, нам следует упомянуть, что компилятор Java отмечает «недостижимые» операторы как ошибки во время компиляции. Компилятор определяет недостижимый оператор, как оператор, который вообще

не будет вызываться. Конечно, многие методы могут никогда не быть вызваны в вашем коде, но компилятор обнаруживает только те, которые он может «подтвердить» как никогда не используемые при простой проверке во время компиляции.

Например, метод с безусловным оператором возврата посреди компиляции вызывает ошибку компиляции, как и метод с условным оператором, о котором компилятор может сказать как об операторе, который никогда не выполняется:

```
f (1 < 2)
return;
// недостижимый оператор
```

Выражения

Выражение производит результат или значение, когда вычисляется. Значение выражения может быть числового типа, как в арифметическом выражении; ссылочного типа, как в локализации объекта; или специального типа `void` — это декларированный тип для метода, который не возвращает значение.

В последнем случае выражение вычисляется только по его *побочным эффектам*; то есть та работа, которую оно выполняет помимо произведения значения. Тип выражения известен во время компиляции. Значение, производимое во время выполнения, либо этого типа, либо, в случае ссылочного типа, сходного (присваиваемого) подтипа.

Операции

Java поддерживает почти все стандартные операции языка C. Эти операции в Java имеют тот же приоритет, что и в C, как показано в табл. 4.3.

Мы также должны отметить, что процентная операция (%) не строго является модулем, а остатком и может иметь отрицательное значение.

В Java также добавилось несколько новых операций. Как мы видели, операция + может использоваться со значениями типа `String` для выполнения конкатенации строк. Поскольку все целочисленные типы в Java являются знаковыми значениями, операция >> может использоваться для совершения арифметического сдвига вправо со знаковыми

расширениями. Операция >>> воспринимает операнд как беззнаковое число и производит арифметический сдвиг вправо без расширения знака. Операция new используется для создания объектов; вскоре мы рассмотрим это в подробностях.

Табл. 4.3. Операции Java

Степень важности	Операция	Тип операнда	Описание
1	++, –	Арифметический	Инкремент и декремент
1	+, -	Арифметический	Одинарный плюс и минус
1	~	Целочисленный	Поразрядное отрицание
1	!	Булевый	Логическое отрицание
1	(*тип*)	Любой	Преобразование
2	*, /, %	Арифметический	Умножение, деление, остаток
3	+, -	Арифметический	Сложение и вычитание
3	+	Строчный	Конкатенация строк
4	<<	Целочисленный	Сдвиг влево
4	>>	Целочисленный	Сдвиг вправо со знаковым разрядом
4	>>>	Целочисленный	Сдвиг вправо без знакового разряда
5	<, <=, >, >=	Арифметический	Сравнение чисел
5	instanceof	Объект	Сравнение типов
6	==, !=	Простой	Равенство и неравенство значения
6	==, !=	Объект	Равенство и неравенство ссылки
7	&	Целочисленный	
7	&	Булевый	
8	^	Целочисленный	Побитовое исключающее или (XOR)
8	^	Булевый	Булево исключающее или
9	|	Целочисленный	Побитовое или
9	|	Булевый	Булево или
10	&&	Булевый	Условное и
11	||	Булевый	Условное или
12	?:	Нет данных	Условная тернарная операция
13	=	Любой	Назначение

Назначение

Хотя инициализация переменной (то есть объявление и назначение вместе) рассматривается как оператор без результативного значения, само назначение переменной является выражением:

```
int i, j; //оператор
i = 5; //и выражение, и оператор
```

Обычно мы полагаемся на назначение только за его побочные эффекты, но назначение может быть использовано как значение в другой части приложения:

```
j = ( i = 5 );
```

Опять же, если сильно полагаться на порядок вычислений (в этом случае используя сложное присвоение в сложных выражениях), то можно сделать код трудным для понимания и тяжелым для прочтения.

Значение `null`

Выражение `null` может быть назначено любому ссылочному типу. Оно означает «нет ссылки». Ссылка `null` не может сослаться ни на что и попытка сослаться выдаст исключение `NullPointerException` во время выполнения.

Доступ к переменным

Операция точка (`.`) используется для выбора членов класса или экземпляра объекта (мы поговорим об этом в деталях в следующих главах). Он может возвращать значение переменной экземпляра (объекта) или статической переменной (класса). Он также может уточнять метод, который будет вызываться для объекта или класса:

```
int i = myObject.length;
String s = myObject.name;
myObject.someMethod();
```

Выражение ссылочного типа может использоваться в сложных вычислениях при выборе дальнейших переменных или методов по результату:

```
int len = myObject.name.length();
int initialLen = myObject.name.substring(5, 10).length();
```

Здесь мы вычислили длину нашей переменной `name`, вызвав метод `length()` объекта `String`. Во втором случае мы совершили промежуточное действие и затребовали подстроку сроки `name`. Метод подстроки класса `String` также возвращает ссылку `String`, для которой мы запрашиваем ширину. Объединение операций таким образом также называется *сцепленным* вызовом методов (упомянем его позже). Одной из сцепленных операций выбора, которую мы уже многократно использовали, является вызов метода `println()` для переменной из класса `System`:

```
System.out.println("вызов println ");
```

Вызов метода

Методы являются функциями, которые находятся внутри класса и могут быть доступны через класс или его экземпляры в зависимости от типа метода. Вызов метода означает выполнение операторов его тела, передавая любые необходимые переменные параметра и по возможности получая значение взамен. Вызов метода — это выражение, которое приводит к результату. Тип значения является *возвращаемым типом функции* метода:

```
System.out.println( "Hello, World..." );
int myLength = myString.length();
```

Здесь мы вызвали методы `println()` и `length()` для разных объектов. Метод `length()` возвращает целое число; возвращаемым типом `println()` является `void` (нет значения).

Это довольно просто, но в главе 5 мы увидим, что все становится немного сложнее, когда у методов одинаковые имена, но разные типы параметров в одном классе или когда метод переопределяется в дочернем классе, как описано в главе 6.

Создание объекта

Объекты в Java назначаются операцией `new`:

```
Object o = new Object();
```

Аргумент операции `new` является конструктором класса. *Конструктор* — это метод, у которого всегда такое же имя, как и у класса. Конструктор указывает любые необходимые параметры, чтобы создать экземпляр объекта. Значением выражения `new` является ссылка типа

созданного объекта. У объектов всегда есть один или более конструкторов, хотя они не всегда могут быть доступными для вас.

Мы рассмотрим создание объекта в подробностях в главе 5. Сейчас просто отметьте, что создание объекта — это тип выражения, а результат — ссылка на объект. Небольшой особенностью является то, что связь в операции `new` «теснее», чем в селекторе-точке (`.`). Поэтому вы можете создать новый объект и вызвать метод в нем, не назначая объект переменной ссылочного типа, если у вас есть для этого причина:

```
int hours = new Date().getHours();
```

Класс `Date` является утилитным классом, который представляет текущее время. Здесь мы создаем новый экземпляр `Date` при помощи операции `new` и вызываем его метод `getHours()`, чтобы получить текущий час в виде целого числа. Ссылка на объект `Date` существует достаточно долго, чтобы служить вызову метода, а затем освобождается и отправляется в мусор в какой-то момент в будущем (смотрите главу 5 для получения информации о чистке памяти).

Вызов методов в ссылках на объекты таким путем является опять же вопросом стиля. Определенно, более понятный способ — назначить вспомогательную переменную типа `Date` для хранения нового объекта, а затем вызвать метод `getHours()`. Однако объединение операций таким образом является распространенным.

Операция `instanceof`

Операция `instanceof` может использоваться для определения типа объекта во время выполнения. Она проверяет, является ли объект таким же типом или подтипом целевого типа. Это то же самое, что и узнать, может ли объект быть назначен переменной целевого типа. Целевой тип может быть классом, интерфейсом или типом массива, как мы увидим позже. `instanceof` возвращает булево значение, которое показывает, совпадает ли объект с типом:

```
Boolean b;
String str = "foo";
b = ( str instanceof String ); // истина, str является String
b = ( str instanceof Object ); // также истина, String is an Object
//b = ( str instanceof Date ); // Компилятор достаточно умен, чтобы
уловить это!
```

`Instanceof` также верно сообщает, является ли объект типом массива или указанным интерфейсом (об этом поговорим позже).

```
if ( foo instanceof byte[] )
...
```

Следует отметить, что значение `null` нельзя считать экземпляром какого-либо объекта. Следующая проверка возвращает `false` вне зависимости от типа переменной:

```
String s = null;
if ( s instanceof String )
// false, null не является экземпляром чего-либо
```

Исключения

У Java есть свои корни во встроенных системах — программном обеспечении, которое работает в специализированных устройствах, таких как карманные компьютеры, мобильные телефоны и продвинутые тостеры. В таких видах приложений особенно важно, чтобы ошибки программного обеспечения обрабатывались хорошо. Большинство пользователей согласится, что это неприемлемо, когда телефон просто выходит из строя или тостер (и, возможно, дом) сгорает из-за сбоя в программном обеспечении. Если предположить, что мы не можем исключить возможность ошибок в ПО, то методологическое распознание и работа с гипотетическими ошибками на уровне приложения будут шагом в правильном направлении. За работу с ошибками в некоторых языках полностью отвечает программист. Сам язык не предоставляет никакой помощи в определении типов ошибок и никаких инструментов для того, чтобы легко с ними справиться. В языке C рутинная операция обычно указывает на ошибку, возвращая «необоснованное» значение (например, идиоматическое — 1 или `null`). Как программист вы должны знать, что формирует плохой результат и что он означает. Часто бывает неудобно обходить ограничения передачи значений ошибок в нормальный путь потока данных*. Еще худшей проблемой является то, что определенные типы ошибок могут обоснованно возникнуть почти везде, и невозможно и неразумно в прямой форме тестировать их по всем пунктам в программном обеспечении.

---

* Немного запутанные операторы `setjmp()` и `longjmp()` в языке C могут сохранить положение выполнения кода, а затем вернуться к нему безусловно из находящегося в далекой глубине места. В ограниченном смысле — это функциональность исключений в Java.

Java предлагает элегантное решение этих проблем путем *исключений*. (Обработка исключений в Java похожа на обработку исключений в C++, но не совсем.) *Исключение* указывает на необычное условие или условие ошибки. Контроль программы в безусловном порядке перемещается, или «перебрасывается», в специально указанную часть кода, где он отлавливается и обрабатывается. В этом отношении обработка ошибок ортогональна (или независима) по отношению к нормальному потоку программы. Нам не нужно иметь специальные возвратные значения для всех наших методов; ошибки обрабатываются отдельным механизмом. Управление может быть передано на большое расстояние от глубоко расположенной рутинной программы и обрабатывается в одном месте, где это необходимо, или ошибка может обрабатываться немедленно в своем источнике. Несколько стандартных методов API Java до сих пор возвращают –1 как специальное значение, но они обычно ограничены ситуациями, где мы ожидаем специальное значение или где ситуация в действительности не выходит за пределы*.

Метод Java необходим для определения исключений, которые он может выбросить (например, те, которые он сам не отлавливает), а компилятор следит за тем, чтобы операторы, вызывающие метод, обрабатывали их. Таким образом, информация о том, какие ошибки может произвести метод, продвигается на тот же уровень важности, что и его аргумент, и типы возврата. Вы все еще можете решить игнорировать очевидные ошибки, но в Java вы должны делать это явно. (Мы обсудим «исключения в выполнении», которые не обязательно должны быть объявлены или обработаны методом в один миг).

Исключения и классы ошибок

Исключения представлены экземплярами класса `java.lang.Exception` и его подклассами. Подклассы `Exception` могут содержать специализированную информацию (и, возможно, поведение) для разных видов исключительных условий. Однако чаще они являются просто «логическими» подклассами, которые служат только для определения нового типа исключения. Рис. 4.1 показывает подклассы класса `Exception` в пакете `java.lang`. Это должно дать вам понятие о том, как организованы исключения. Большинство других пакетов определяют

* Например, метод `getHeight()` класса `Image` возвращает `-1`, если высота еще не известна. Ошибка не возникла; высота будет доступна в будущем. В этой ситуации создание исключения будет неуместно.

собственные типы исключения, которые обычно являются подклассами самого Exception или его важного подкласса RuntimeException, до которого мы доберемся через мгновение.

Например, важным классом исключения является IOException в пакете java.io. Класс IOException расширяет Exception и имеет много подклассов для типичных проблем ввода/вывода (таких как FileNotFoundException) и проблем сети (таких как MalformedURLException). Сетевые исключения относятся к пакету java.net. Еще один важный потомок IOException — RemoteException, который относится к пакету java.rmi. Он используется, когда возникают проблемы во время удаленного вызова методов (RMI). В этой книге мы упоминаем исключения, о которых вам нужно знать, по мере того, как мы сталкиваемся с ними.

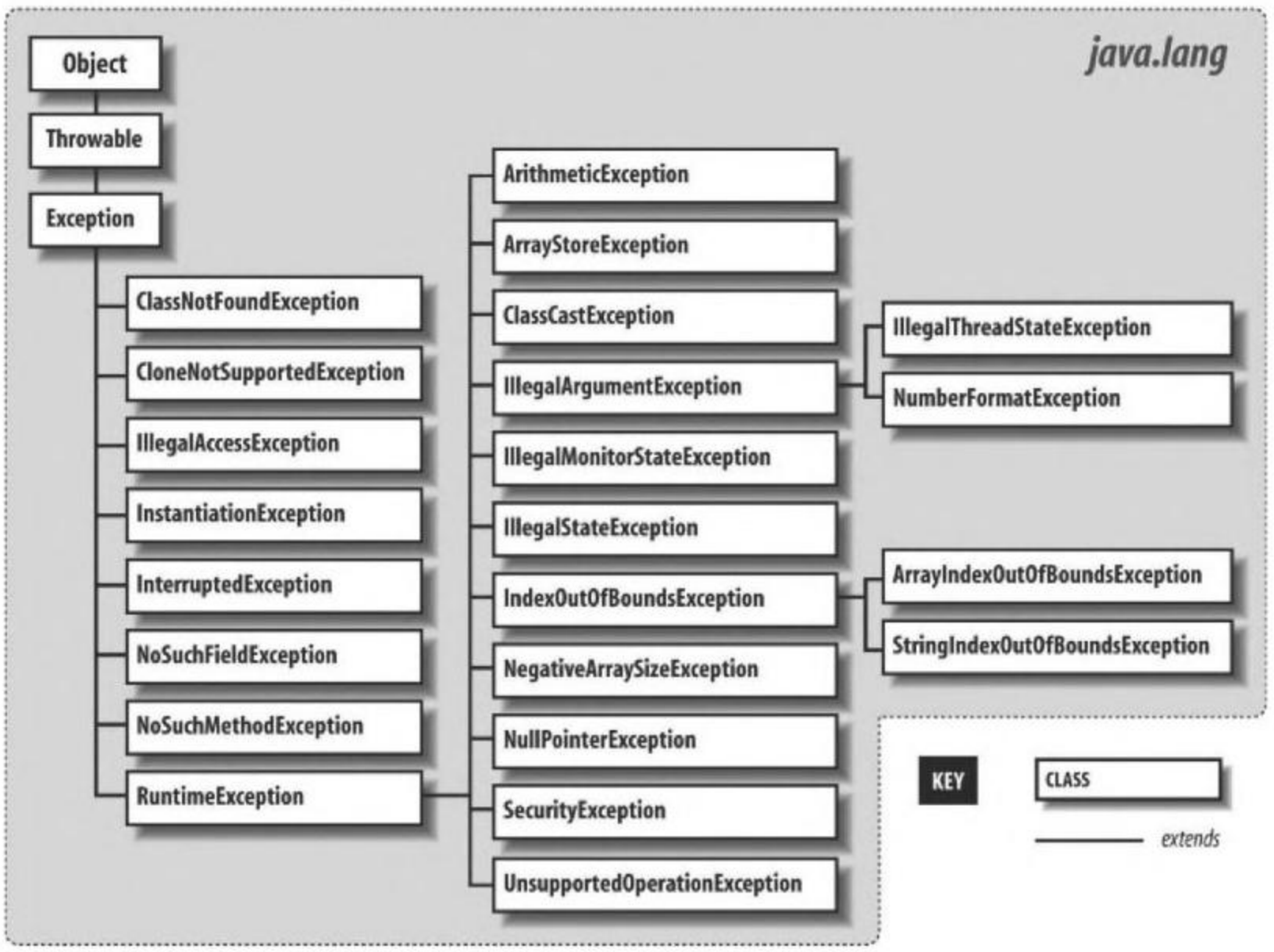

Рис. 4.1. Подклассы java.lang.Exception

Объект класса Exception создается кодом в том месте, где возникает условие ошибки. Он может быть разработан, чтобы содержать любую необходимую информацию для описания исключительного условия, а также включает *трессировку стека* для отладки. (Трессировка стека — это список всех вызываемых методов и порядок, в котором они вызыва-

ются для достижения места, где возникло исключение.) Объект класса `Exception` передается в качестве аргумента обрабатывающему блоку кода вместе с потоком управляющей информации. Вот откуда происходят понятия *выдавать* и *отлавливать*: объект класса `Exception` выдается одним местом в коде и отлавливается другим, где выполнение продолжается.

API Java также определяет класс `java.lang.Error` по неустранимым ошибкам. Подклассы `Error` в пакете `java.lang` показаны на рис. 4.2. Выдающийся тип класса `Error` — `AssertionError`, который используется оператором `assert` для указания неисправности (операторы подтверждения отсутствия ошибок будут обсуждаться позже в этой главе).

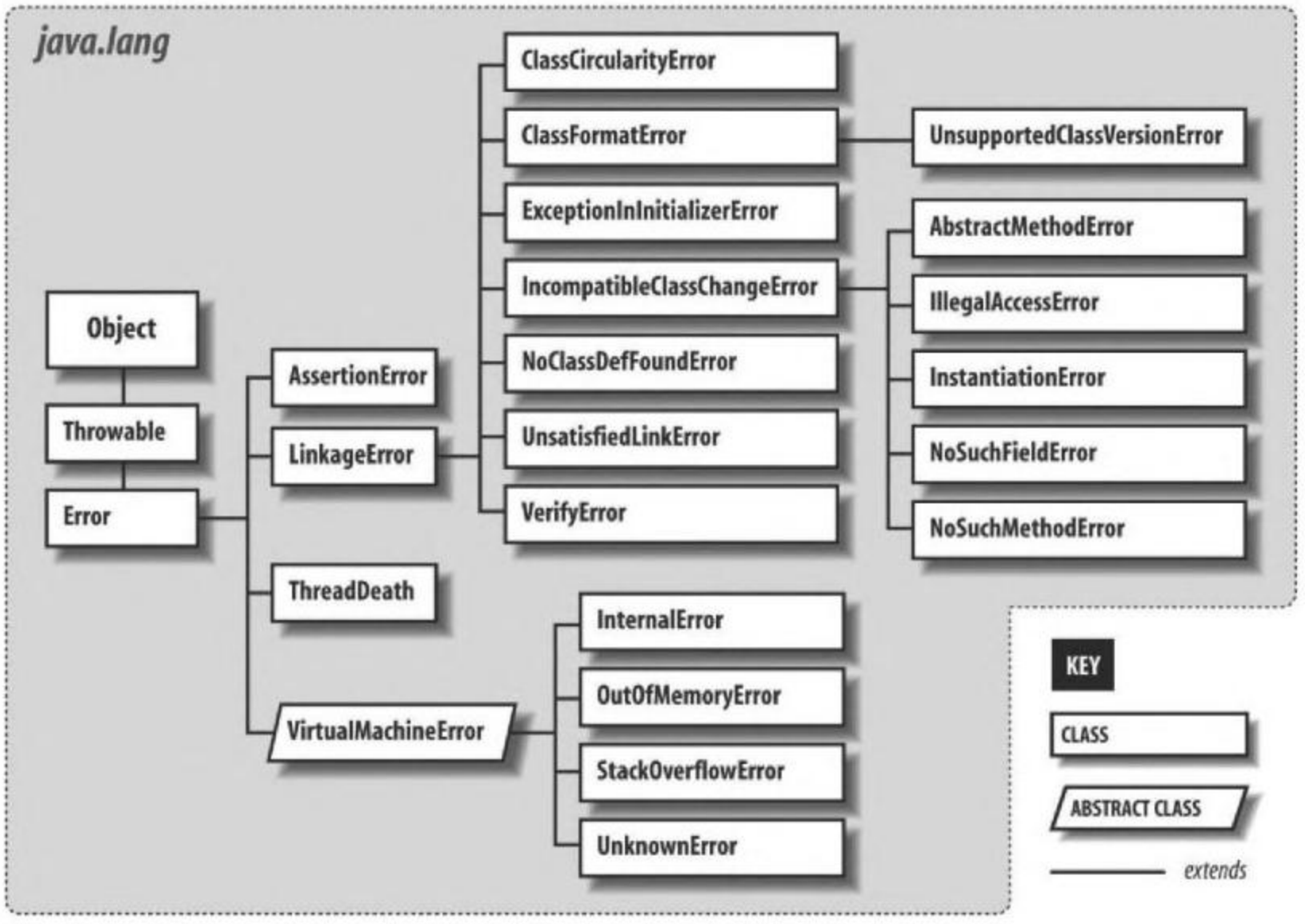

Рис. 4.2. Подклассы `java.lang.Error`

Несколько других пакетов определяют их собственные подклассы `Error`, но подклассы `Error` намного менее распространены (и реже употребляются), чем подклассы `Exception`. В основном вам нет необходимости беспокоиться об этих ошибках в вашем коде (то есть вам не нужно их отлавливать); они предназначены для указания неустранимых проблем или ошибок виртуальной машины. Ошибка такого рода обычно заставляет интерпретатор показывать сообщение и выходить. Вам активно препятствуют в попытках отловить их или избавиться от

них, потому что предполагается, что они указывают на неустранимую ошибку программы, а не состояния рутинной программы.

Классы `Exception` и `Error` являются подклассами `Throwable`. Класс `Throwable` является основным классом для объектов, которые могут быть «выданы» оператором `throw`. В основном вам нужно расширять только `Exception`, `Error` или один из их подклассов.

Обработка исключений

Предохраняющие операторы `try/catch` оборачивают блок кода и отлавливают назначенные типы исключений, которые возникают в нем:

```
try {
readFromFile("foo");
...
}
catch ( Exception e ) {
// обработка ошибки
System.out.println( "Исключение при прочтении файла: " + e );
...
}
```

В этом примере исключения, которые возникают в теле части оператора `try`, направлены на пункт `catch` для возможной обработки. Пункт `catch` действует как метод; он указывает в качестве аргумента тип исключения, который хочет обработать и если он вызывается, он получает объект `Exception` в качестве аргумента. Здесь мы получаем объект в переменной `e` и выводим его вместе с сообщением.

Оператор `try` может иметь несколько операторов `catch`, которые определяют разные типы `Exception` (подклассы):

```
try {
readFromFile("foo");
...
}
catch ( FileNotFoundException e ) {
// Обработать "файл не найден"
...
}
catch ( IOException e ) {
```

```
// Обработать ошибку чтения
...
}
catch ( Exception e ) {
// Обработать все другие ошибки
...
}
```

Операторы `catch` вычисляются по очереди, и берется первое присваиваемое совпадение. В лучшем случае выполняется один оператор `catch`, что означает, что исключения должны быть перечислены от более конкретного к менее конкретному. В предыдущем примере мы предусматриваем, что гипотетический метод `readFromFile()` может выбросить два разных типа исключений: одно для ненайденного файла и второе для более общей ошибки чтения. В предшествующем примере исключения `FileNotFoundException` является подклассом `IOException`, поэтому если бы первого оператора `catch` не было там, исключение было бы отловлено вторым. Подобным образом любой подкласс `Exception` присваиваем родительскому типу `Exception`, поэтому третий оператор `catch` отловит то, что передано предыдущими двумя. Здесь он действует как оператор `default` в операторе `switch` и обрабатывает любые оставшиеся варианты. Мы показали это здесь для полноты картины, но обычно вы постараетесь быть как можно более точными в типах исключений, которые вам нужно будет отловить.

Одна из прелестей схемы `try/catch` — любой оператор в блоке `try` может предполагать, что все предыдущие операторы в блоке выполнились. Проблема не возникнет внезапно из-за того, что программист забыл проверить возвратное значение метода. Если предыдущий оператор не сработает, выполнение немедленно перейдет к оператору `clause`; следующие операторы никогда не выполняются.

В Java 7 есть альтернатива использованию множества операторов `catch` для обработки многих типов дискретных исключений в одном операторе `catch` с использованием символа `|` или следующего синтаксиса:

```
try {
// прочесть из сети...
// записать в файл..
catch ( ZipException | SSLException e ) {
logException( e );
}
```

Используя этот синтаксис или символ |, мы получаем оба типа исключений в одном и том же операторе catch. Итак, каким же является действительный тип переменной e, которую мы передаем нашему методу log? (Что мы можем с этим сделать?) В данном случае это будет не исключение ZipException и не SSLException, а IOException, который является ближайшим предком этих двух исключений (ближайший тип родительского класса, к которому они оба присваиваются). Во многих случаях ближайшим общим типом для двух или более типов аргументов исключения может быть Exception, родитель всех типов исключений. Разница между отловом этих типов дискретных исключений при помощи многовариантного catch и упорядочиванием ваших операторов catch от более специфического типа до более распространенного («широкого» к «узкому») дает вам большую гибкость для структуризации ваших операторов catch с целью объединить логику обработки, где это необходимо, и избежать повторения кода. Существует больше нюансов данного свойства, и мы вернемся к этому после обсуждения «выброса» и «повторного выброса» исключений.

Восхождение исключений

Что если мы не отловили исключение? Куда оно денется? Что ж, если нет огораживающего оператора try/catch, исключение всплывет из метода, из которого оно возникло, и выбрасывается из этого метода оператору, который его вызывает. Если этот пункт вызывающего метода находится внутри оператора try, управление переходит к соответствующему оператору catch. В противном случае исключение продолжает распространяться по стеку вызова от одного метода до оператора, который его вызывает. В таком случае исключение всплывает, пока оно не будет отловлено или пока не выскочит из верхней части программы, прерывая ее сообщением об ошибке в выполнении. Есть кое-что еще в данном случае, компилятор может принудить нас иметь с этим дело в рабочем порядке, но мы вскоре вернемся к этому.

Давайте посмотрим на другой пример. На рис. 4.3 метод getContent() вызывает метод openConnection() в оператора try/catch. В свою очередь openConnection() вызывает метод sendRequest(), который вызывает метод write(), чтобы отправить некие данные.

На этом рисунке второй вызов write() выбрасывает исключение IOException. Поскольку метод sendRequest() не содержит оператор try/catch для обработки исключения, оно выбрасывается снова из того места, где оно было вызвано в методе openConnection(). Поскольку

openConnection() также не отлавливает исключение, оно выбрасывается снова. В конце концов, оно отлавливается оператором try в методе getContent() и обрабатывается его оператором catch. Обратите внимание: каждый выбрасывающий метод должен объявлять, что он может выбросить определенный тип исключения при помощи оператора throws. Мы вскоре обсудим это.

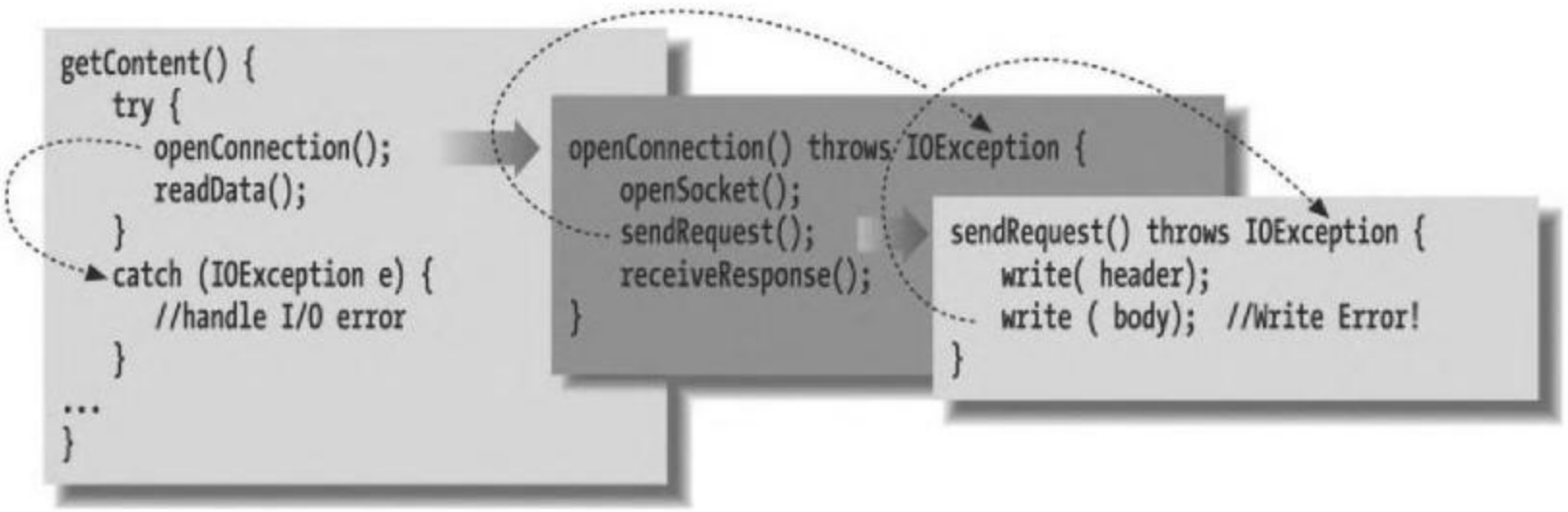

Рис. 4.3. Распространение исключения

Трассировка стека

Поскольку исключение может всплыть на некоем расстоянии перед тем, как оно отловлено и отработано, нам нужен способ определить точно, где оно может быть выброшено. Также очень важно знать контекст того, как был достигнут путь исключения; то есть, какие методы вызвали какие методы, чтобы добраться до этого пункта. Для таких типов отладки и регистрации все исключения могут выводиться в трассировку стека, которая перечисляет методы их происхождения и все встроенные вызовы методов, которые помогли попасть им туда. Чаще всего пользователь видит трассировку стека, когда она выводится при использовании метода printStackTrace().

```
try {
// сложное, глубоко расположенное задание
} catch ( Exception e ) {
// сброс информации о том, где точно возникло исключение
e.printStackTrace( System.err );
...
}
```

Например, трассировка стека на исключение может выглядеть таким образом:

```
java.io.FileNotFoundException: myfile.xml
at java.io.FileInputStream.<init>(FileInputStream.java)
at java.io.FileInputStream.<init>(FileInputStream.java)
at MyApplication.loadFile(MyApplication.java:137)
at MyApplication.main(MyApplication.java:5)
```

Эта трассировка стека указывает, что метод `main()` класса `MyApplication` вызвал метод `loadFile()`. Метод `loadFile()` затем пытался сконструировать `FileInput Stream`, который выбросил исключение `FileNotFoundException`. Отметьте, что когда трассировка стека достигает системы классов Java (такой как `FileInputStream`), номера строк могут потеряться. Это также может случиться, когда код оптимизируется некими виртуальными машинами. Обычно есть способ временно отключить оптимизацию, чтобы найти точные номера строк. Однако в сложных ситуациях изменение расчета времени приложения может влиять на проблему, которую вы пытаетесь отладить, также могут понадобиться другие техники отладки.

Методы, применимые к исключению, позволяют вам получить информацию о трассировке стека программным путем также при использовании метода `getStackTrace()` класса `Throwable`. (`Throwable` является базовым классом `Exception` и `Error`.) Этот метод возвращает массив объектов `StackTraceElement`, каждый из которых представляет вызов метода в стеке. Вы можете спросить `StackTraceElement` о деталях локализации метода, используя методы `getFileName()`, `getClassName()`, `getMethodName()` и `getLineNumber()`. Нолевой элемент массива находится вверху стека, последняя строка, которая вызывает исключение; последующие элементы возвращаются на один вызов метода, пока не будет достигнут исходный метод `main()`.

Проверяемые и непроверяемые исключения

Мы упоминали ранее, что Java заставляет нас быть точными в обработке ошибок, но нет необходимости требовать, чтобы каждый мыслимый тип ошибки обрабатывался точно в каждой ситуации. Исключения в Java таким образом разделяются на две категории: *проверяемые* и *непроверяемые*. Большинство исключений на уровне приложения являются проверяемыми, это означает, что каждый метод, который выбрасывает исключение либо генерируя его самостоятельно (это мы обсудим позже), либо игнорируя то исключение, которое возникло в нем, должен объявлять, что он может выбросить этот тип исключения в операторе

throws в объявлении метода. Мы еще не обсуждали в подробностях объявление методов (см. главу 5). Сейчас все, что вам необходимо знать, это то, что методы должны объявлять проверяемые исключения, которые они могут передать или позволяют, чтобы они были переданными.

Опять же, как видно из рис. 4.3, методы openConnection() и sendRequest() указывают, что они могут выбросить IOException. Если бы нам нужно было выбросить несколько типов исключений, мы бы могли объявить их через запятую:

```
void readFile( String s ) throws IOException, InterruptedException {
...
}
```

Оператор throws сообщает компилятору, что метод является гипотетическим источником этого типа проверяемого исключения и что любой, кто вызывает этот метод, должен быть готов справиться с ним. Вызывающий оператор затем должен использовать блок try/catch для его обработки, или он в свою очередь должен объявить, что может выбросить исключение от себя. Напротив, исключения, которые представляют собой подклассы классов java.lang.RuntimeException или java.lang.Error, являются непроверяемыми. См. рис. 4.1, чтобы увидеть подклассы RuntimeException. (Подклассы Error в основном зарезервированы для серьезных проблем с загрузкой класса или выполнением.) Это не ошибка во время компиляции, чтобы игнорировать возможность этих исключений; методы также не должны объявлять, что они могут выбросить их. Во всех других отношениях непроверяемые исключения ведут себя так же, как проверяемые исключения. Мы можем отлавливать их, если хотим, но в этом случае у нас нет такой необходимости.

Проверяемые исключения предназначены для решения проблем на уровне приложения, таких как отсутствующие файлы и недоступные хосты. Как хорошие программисты (и добропорядочные граждане) мы должны разрабатывать программное обеспечение так, чтобы с изяществом выходить из подобных ситуаций. Непроверяемые исключения предназначены для проблем системного уровня, таких как «недостаточно памяти» и «индекс массива за пределами границ». Поскольку это может указывать на программные ошибки на уровне приложения, они могут возникнуть почти везде, и обычно невозможно избавиться от них. К счастью, они являются непроверяемыми исключениями и вам не нужно оборачивать каждую операцию с массивом и индексом в оператор try/catch (или объявлять все вызывающие методы как потенциальный их источник).

Таким образом, проверяемые исключения — это проблемы, с которыми грамотное приложение должно по возможности достойно справляться; непроверяемые исключения (исключения выполнения или ошибки) — это проблемы, после которых мы не ожидаем восстановления программного обеспечения.

Типы ошибок точно предназначены, чтобы быть условиями, с которыми мы обычно не должны пытаться справиться или восстановиться.

Выброс исключений

Мы можем выбрасывать наши собственные исключения — или экземпляры класса `Exception`, одного из его существующих подклассов, или наши собственные специализированные классы исключений. Все, что нам нужно сделать, — это создать экземпляр `Exception` и выбросить его оператором `throw`:

```
throw new IOException();
```

Выполнение останавливается или передается ближайшему ограничивающему оператору `try/catch`, который может обработать тип исключения. (Мало смысла в хранении ссылки к объекту `Exception`, который мы создали здесь.) Альтернативный конструктор позволяет нам указывать строку с сообщением об ошибке:

```
throw new IOException("Солнечные пятна!");
```

Вы можете получить эту строку, используя метод `getMessage()` объекта `Exception`. Хотя часто можно просто вывести сам объект исключения (или `toString()`), чтобы получить сообщение и трассировку стека.

Обычно все типы `Exception` имеют такой конструктор `String`. Предшествующее сообщение `String` не очень полезно. Обычно он выбрасывает более специфический подкласс `Exception`, который отлавливает подробности или хотя бы более специфические строчные объяснения. Вот еще один пример:

```
public void checkRead( String s ) {
if ( new File(s).isAbsolute() || (s.indexOf("..") != -1) )
throw new SecurityException(
"В доступе к файлу : "+ s +" отказано.");
}
```

В этом коде мы частично применяем метод для проверки некорректных путей. Если мы находим таковой, мы выбрасываем `SecurityException` с некоторой информацией о нарушении.

Конечно, мы можем включить любую другую информацию, которая будет полезна в наших собственных специализированных подклассах `Exception`. Хотя часто просто получить новый тип исключения — уже хорошо, потому что его достаточно, чтобы помочь направить поток управления. Например, если мы создаем парсер, мы, возможно, захотим создать собственный тип исключения, чтобы указать определенный тип ошибки:

```
class ParseException extends Exception {
ParseException() {
super();
}
ParseException( String desc ) {
super( desc );
}
}
```

См. главу 5 для полного описания классов и конструкторов классов. Тело нашего класса `Exception` здесь просто позволяет создать `ParseException` удобными способами, которыми мы ранее создавали исключения (либо по образцу, либо просто строчным описанием). Теперь, когда у нас есть наш новый тип описания, мы можем ограничиваться следующим:

```
// Где-то в нашем коде
...
try {
parseStream( input );
} catch ( ParseException pe ) {
// Плохой ввод…
} catch ( IOException ioe ) {
// Проблемы коммуникации низкого уровня
}
```

Как видите, хотя наше новое исключение в данный момент не содержит никакой специализированной информации о проблеме (хотя определенно могло бы), оно действительно позволит нам отличать ошибку разбора от произвольной ошибки ввода/вывода в том же самом куске кода.

Связывание и повторный выброс исключений

Иногда вам захочется принять меры, основанные на исключении, а затем вернуться и выбросить новое исключение на его место. Это распространено при создании фреймворков, где детализированные исключения низкого уровня обрабатываются и представлены исключениями более высокого уровня, которыми нужно управлять более просто. Например, вы можете захотеть отловить исключение `IOException` в пакете коммуникаций, возможно, произвести некую чистку и, в конце концов, выбросить свое исключение высокого уровня, может, что-то наподобие `LostServerConnection`.

Вы можете сделать это очевидным путем, просто отловив исключение, а затем выбросив новое, но в таком случае вы потеряете важную информацию, включая трассировку стека оригинального «причинного» исключения. Чтобы разобраться с этим, вы можете использовать технику *связывания исключений*. Это означает, что вы включаете причинное исключение в новое исключение, которое выбрасываете. Java имеет эксплицитную поддержку связывания исключений. Основной класс `Exception` может быть сконструирован с исключением в качестве аргумента или стандартного сообщения типа `String` и исключения:

```
throw new Exception( "Вот эта история...", causalException );
```

Вы можете получить доступ к обернутому исключению позже при помощи метода `getCause()`. Что более важно, Java автоматически выводит оба исключения и их соответствующие трассировки стека, если вы выведете исключение или если оно показано пользователю.

Вы можете добавить этот тип конструктора вашим собственным подклассам исключений (делегируя родительскому конструктору) или можете использовать преимущество этого образца, используя метод `initCause()` класса `Throwable`, чтобы установить причинное исключение эксплицитно после конструирования вашего исключения и перед выбросом его:

```
try {
// ...
} catch ( IOException cause ) {
Exception e =
new IOException("Что мы здесь имеем — это ошибка в коммуникации…");
e.initCause( cause );
throw e;
}
```

Иногда достаточно просто совершить некую регистрацию или принять какие-то меры, а затем заново выбросить исходное исключение:

```
try {
// ...
} catch ( IOException cause ) {
log( e ); // записать это
throw e; // заново выбросить это
}
```

Но примите к сведению, что, если вы это сделаете, трассировка стека, включенная в исключение, покажет новое положение выброса как исходное.

Ограничение повторного выброса исключений

До версии Java 7, если вы хотели обработать множество типов исключений одним оператором `catch`, а затем снова выбросить исходное исключение, вы неизбежно придете к расширению объявленного типа исключения до того, что необходимо, чтобы захватить их, или вам придется проделать большую работу, чтобы избежать этого. В Java 7 компилятор стал умнее и сейчас может делать основную часть работы за нас, позволив нам в большинстве случаев ограничивать набор генерируемых исключений исходными типами. Это лучше всего объясняется примером:

```
void myMethod() throws ZipException, SSLException
{
try {
// Возможно, из-за ZipException или SSLException
} catch ( Exception e ) {
log( e );
throw e;
}
```

В этом примере мы ленимся и просто отлавливаем все исключения при помощи всеобъемлющего оператора отлова `Exception`, чтобы зарегистрировать их до повторного выброса. До Java 7 компилятор настаивал, чтобы оператор `throws` нашего метода объявил, что выбрасывает широкий тип `Exception`. Однако компилятор Java сейчас достаточно умен в большинстве случаев, чтобы анализировать актуальные типы исключений, которые могут быть выброшены и позволяют нам назначить точный набор типов. То же самое было бы справедливо, если бы

мы использовали многотипный оператор `catch` в этом примере, как вы могли догадаться. Вышеописанное немного менее интуитивно, но очень полезно в укреплении специфичности обработки ошибок в коде, включая код, написанный до Java 7, не требующий потенциально мудреной переработки операторов `catch`.

Разумное использование оператора try

Оператор `try` возлагает условие на оператор, который его ограничивает. Он говорит, что если исключение возникает внутри него, остальные операторы запрещены. Это имеет последствия для инициализации локальной переменной. Если компилятор не может определить, произойдет присваивание переменной, помещенной в блок `try/catch`, он не позволит нам использовать переменную. Например:

```
void myMethod() {
int foo;
try {
foo = getResults();
}
catch ( Exception e ) {
...
}
int bar = foo; // Ошибка во время компиляции: возможно foo не был
инициализирован
```

В этом примере мы не можем использовать `foo` в указанном месте, поскольку есть вероятность того, что ему не было назначено значение. Одна очевидная причина — это поместить назначение внутрь оператора `try`:

```
try {
foo = getResults();
int bar = foo; // Нормально, поскольку мы попадаем сюда, только
// если предыдущее назначение выполнилось
}
catch ( Exception e ) {
...
}
```

Иногда это работает просто отлично. Однако теперь у нас есть такая же проблема, если мы хотим использовать панель позже в `myMethod()`.

Если мы не будем осторожны, мы можем закончить тем, что будем запихивать все в оператор `try`. Однако ситуация меняется, если мы переместим управление из метода в операторе `catch`:

```
try {
foo = getResults();
}
catch ( Exception e ) {
...
return;
}
int bar = foo; //Нормально, поскольку мы попадаем сюда, только
// если предыдущее назначение выполнилось
```

Компилятор достаточно умен, чтобы понять, что, если бы ошибка возникла в операторе `try`, мы бы не достигли панели назначения, поэтому он позволяет нам сослаться на `foo`. Ваш код будет диктовать свои собственные нужды; вы просто должны быть осведомлены об опциях.

Оператор `finally`

Что если нам нужно сделать что-то важное перед тем, как мы покинем метод из одного из операторов `catch`? Чтобы избежать дублирования кода в каждой ветви `catch` и сделать чистку более точной, вы можете использовать оператор `finally`. Оператор `finally` может быть добавлен после `try` и другого связанного оператора `catch`. Любые операторы в теле оператора `final` гарантированно исполнятся вне зависимости от того, как управление оставит тело `try`, было ли выброшено исключение или нет:

```
try {
// Сделать что-то здесь
}
catch ( FileNotFoundException e ) {
...
}
catch ( IOException e ) {
...
}
catch ( Exception e ) {
...
```

```
}
finally {
// Здесь всегда производится чистка
}
```

В этом примере операторы в конце концов выполняются в момент чистки, вне зависимости от того, как управление покидает `try`. Если управление переходит к одному из операторов `catch`, операторы в `finally` выполняются, когда завершится `catch`. Если ни один из операторов `catch` не справится с управлением, операторы `finally` выполнятся перед тем, как исключение распространится на следующий уровень.

Если операторы в `try` выполнятся чисто или если мы произведем `return`, `break` или `continue`, операторы в `finally` также будут выполняться. Чтобы гарантировать, что некоторые операторы будут выполняться, мы также можем использовать `try` и `finally` без операторов `catch`:

```
try {
// Выполнить что-то здесь
return;
}
finally {
System.out.println("Whoo-hoo!");
}
```

Исключения, которые возникают в операторах `catch` или `finally` обрабатываются нормально; поиск ограничивающего `try/catch` начинается снаружи неисправного оператора `try`, после того как выполнился `finally`.

Оператор `try` с очисткой ресурсов

Оператор `finally` часто используется, чтобы обеспечить приведение в порядок ресурсов, используемых в операторе `try`, вне зависимости от того, как код покидает блок.

```
try {
// Socket sock = new Socket(...);
// работа с sock
} catch( IOException e ) {
...
}
```

```
finally {
if ( sock != null ) { sock.close(); }
}
```

Здесь мы подразумеваем под приведением в порядок освобождение от затратных ресурсов или близких связей, таких как файлы, сокеты или связи баз данных. В некоторых случаях эти ресурсы могут очиститься сами по себе, когда Java пересмотрит мусор, но это в лучшем случае произойдет неизвестно когда, а в худшем случае не произойдет вовсе или по крайней мере до того, как у вас исчерпаются ресурсы. Поэтому всегда лучше принять меры предосторожности, чтобы избежать таких ситуаций. Существует две проблемы в этом серьезном вопросе: первое — он требует дополнительной работы для выполнения этого фрагмента в вашем коде, включая такие важные вещи, как проверка на ноль, как показано в нашем примере; второе — если вы жонглируете несколькими ресурсами в одном блоке `finally`, есть вероятность, что ваш код чистки выбросит исключение (например, при `close()`) или что работа останется незаконченной.

В Java 7 все очень упростилось благодаря новой форме оператора `try` – «`try` с ресурсами». В этой форме вы можете поместить один или несколько операторов инициализации ресурсов в круглые скобки после ключевого слова `try`, и эти ресурсы автоматически «закроются» для вас, когда управление покинет блок `try`.

```
try (
Socket sock = new Socket("128.252.120.1", 80);
FileWriter file = new FileWriter("foo");
)
{
// работа с сокетом и файлом
} catch ( IOException e ) {
...
}
```

В этом примере мы инициализируем объект `Socket` и `FileWriter` внутри оператора `try`-с-ресурсами и используем внутри тела оператора `try`. Когда проверка покидает оператор `try` либо после успешного выполнения, либо из-за исключения, оба ресурса закрываются автоматически вызовом их метода `close()`. Ресурсы закрываются в *обратном порядке*, относительно того, в котором они были созданы, поэтому зависимости в них приспособлены. Это поведение поддерживается для каждого класса, который реализует интерфейс `AutoCloseable` (что при текущем

подсчете делают 100 разных встроенных классов). Метод `close()` этого интерфейса предназначен для освобождения всех ресурсов, связанных с объектом, и вы можете так же легко применить это в своих собственных классах. При использовании `try` с ресурсами у нас нет необходимости добавлять код, чтобы специально закрыть файл или сокет; это делается для нас автоматически.

Другая проблема, которую решает `try` с ресурсами, — это досадная ситуация, на которую мы ссылаемся, когда исключение может быть выброшено во время операции закрытия. Возвращаясь к предыдущему примеру, где мы использовали оператор `finally`, чтобы произвести нашу чистку, если бы исключение было поднято методом `close()`, оно бы выбросилось в этот момент, полностью запрещая оригинальное исключение из тела оператора `try`. Но в использовании `try` с ресурсами мы сберегаем исходное исключение. Если исключение возникает в теле `try` и одно или более исключений поднимается во время последовательных операций автоматического закрытия, именно исходное исключение тела `try` всплывает в вызывающем операторе. Давайте посмотрим на пример:

```
try (
Socket sock = new Socket("128.252.120.1", 80); // потенциальное исключение
#3
FileWriter file = new FileWriter("foo"); // потенциальное исключение #2
)
{
// работа с сокетом и файлом // потенциальное исключение #1
}
```

Как только `try` начал работу, если исключение возникло как исключение № 1, Java попытается закрыть оба ресурса в обратном порядке, приводя к потенциальным исключениям в местах № 2 и № 3. В этом случае код вызова все еще получит исключение № 1. Исключения № 2 и № 3, однако, не потеряны; они просто «подавлены» и могут быть получены через метод исключения, переданного вызывающему оператору `getSuppressed()` класса `Throwable`. Это возвращает массив всех подавленных исключений.

Проблемы, связанные с производительностью

Благодаря тому, как реализована виртуальная машина Java, предотвращение выброса исключений (при использовании `try`) является бес-

платным. Оно не вызывает дополнительных накладных расходов при выполнении вашего кода. Однако выброс исключения не бесплатен. Когда исключение выбрасывается, Java приходится обнаруживать блок `try/catch` и производить другие продолжительные действия во время выполнения.

Результатом этого является то, что вам следует выбрасывать исключения только в действительно «исключительных» обстоятельствах и избегать использования их в ожидаемых условиях, особенно когда выполнение является проблемой. Например, если у вас цикл, лучше производить небольшое тестирование на каждом проходе и избегать выброса исключений, а не выбрасывать их часто. С другой стороны, если исключение выбрасывается только один раз из миллионов, вам, возможно, захочется устранить расходы тестового кода и не заботиться о расходах на выброс исключения. Общее правило должно заключаться в том, что исключения используются для ситуаций, «выходящих за рамки», или нештатных ситуаций, не рутинных и ожидаемых условий (таких как конец файла).

Утверждения

Утверждение является простым тестом на прохождение/сбой некоего условия, выполняемого во время работы вашего приложения. Утверждения могут использоваться для «санитарной проверки» вашего кода везде, где вы считаете, что определенные условия гарантированы правильным поведением программы. Утверждения отличны от других типов тестов, поскольку они проверяют условия, которые никогда не должны нарушаться на логическом уровне: если утверждение дает сбой, приложение считается испорченным и, как правило, прекращает действие с соответствующим сообщением об ошибке. Утверждения поддерживаются непосредственно языком Java, и они могут включаться или выключаться во время выполнения, чтобы устранить любое снижение производительности при включении их в ваш код.

Использование утверждений для тестирования правильного поведения вашего приложения является простой, но мощной техникой обеспечения качества программного обеспечения. Это заполняет пропасть между теми аспектами программного обеспечения, которые могут быть проверены автоматически компилятором, и теми, которые в основном проверяются «модульным тестированием» или ручным тестированием. Утверждения тестируют предположения о поведении программы и делают их гарантиями (по крайней мере, когда они активированы).

Если вы программировали раньше, вы, должно быть, писали что-то похожее на следующий код:

```
if ( !condition )
throw new AssertionError("фатальная ошибка: 42");
```

Утверждение в Java эквивалентно этому примеру, но выполняется при помощи ключевого слова языка `assert`. Оно принимает булево условие и выборочное значение выражения. Если утверждение не срабатывает, выдается исключение `AssertionError`, что часто приводит к выходу из приложения Java.

Необязательное выражение может вычисляться простым или объектным типом. В любом случае его единственным назначением является превращение в строку и показ пользователю, если утверждение не срабатывает; чаще вы будете прямо использовать строчное сообщение. Вот некоторые примеры:

```
assert false;
assert ( array.length > min );
assert a > 0 : a // Показывает значение a пользователю
assert foo != null : "foo равно null!" // Показывает сообщение "foo равно
null!"
```

В неудавшемся событии первые два утверждения выводят только базовое сообщение, принимая во внимание тот факт, что третье выводит значение a, а последнее выводит сообщение «foo равно null!»

Опять же, важным в утверждениях является не только то, что они более сжаты, чем эквивалентное условие `if`, но то, что они могут быть включены или выключены, когда вы запускаете приложение. Выключение утверждений означает, что их тестовые условия даже не вычисляются, поэтому нет снижения производительности при включении их в ваш код (за исключением, возможно, пробела в файлах классов при их загрузке).

Включение и выключение утверждений

Утверждения включаются или выключаются во время выполнения. При выключении утверждения все еще существуют в файлах классов, но не выполняются и не потребляют времени. Вы можете включать и выключать утверждения для всего приложения или на базе пакетов или классов. По умолчанию утверждения в Java выключены. Чтобы

включить их для вашего кода, используйте консольную метку `-ea` или `-enableassertions`:

```
% java -ea MyApplication
```

Чтобы включить утверждения для определенного класса, добавьте имя класса:

```
% java -ea:com.oreilly.examples.Myclass MyApplication
```

Чтобы включить утверждения только для определенных пакетов, включите имя пакета с последующим многоточием (...):

```
% java -ea:com.oreilly.examples...MyApplication
```

Когда вы включаете утверждения для пакета, Java также включает все подчиненные имена пакета (например, `com.oreilly.examples.text`). Однако вы можете быть избирательными, используя метки `-da` или `-disableassertions`, чтобы отвергнуть индивидуальные пакеты или классы. Вы можете объединить все это, чтобы достигнуть произвольной группировки, наподобие этой:

```
% java -ea:com.oreilly.examples...
-da:com.oreilly.examples.textea:
com.oreilly.examples.text.MonkeyTypewriters
MyApplication
```

Этот пример включает утверждения для всего пакета `com.oreilly.examples`, исключает пакет `com.oreilly.examples.text`, а затем включает исключения только для одного класса `MonkeyTypewriters` в этом пакете.

Использование утверждений

Утверждение приводит в исполнение правило о том, что должно быть неизменно в вашем коде, и в противном случае будет непроверенным. Вы можете использовать утверждение для дополнительной безопасности везде, где хотите проверить ваши предположения о поведении программы, которое не может быть проверено компилятором.

Распространенная ситуация, в которой утверждение остро необходимо, — это проверка на множество условий или значений, где одно всегда должно быть найдено. В этом случае невыполненное утверждение как дефолтное, или «сорвавшееся», поведение указывает на то, что код неиспра-

вен. Например, предположим, что у нас есть значение, называемое direction, которое всегда должно содержать постоянное значение LEFT или RIGHT:

```
if ( direction == LEFT )
doLeft();
else if ( direction == RIGHT )
doRight()
else
assert false : "плохое направление";
```

То же самое применимо к дефолтному оператору case оператора switch:

```
switch ( direction ) {
case LEFT:
doLeft();
break;
case RIGHT:
doRight();
break;
default:
assert false;
}
```

Вообще, вы не должны использовать утверждения для проверки правильности аргументов к методам, потому что вы хотите, чтобы это поведение было частью вашего приложения, а не просто тестом на проверку качества, который может быть выключен. Правильность информации на входе в методе называется его *предусловием*, и вы обычно должны выбрасывать исключения, когда они не совпадают; это возводит предварительные условия к части «контракта» метода с пользователем. Однако проверка правильности результатов вашего метода с утверждениями перед их возвратом является хорошей идеей; это называется *постусловием*.

Иногда определение того, что является, а что не является предусловием, зависит от вашей точки зрения. Например, если метод используется внутри с классом, предусловия могут уже гарантироваться методами, которые вызывают его. Публичные методы класса должны, вероятно, выбрасывать исключения, когда нарушаются их предусловия, но приватный метод может использовать утверждения, поскольку операторы, вызывающие его, всегда являются тесно связанными с кодом, который должен придерживаться правильного поведения.

В итоге отметьте, что утверждения могут тестировать не только простые выражения, но и выполнять сложную проверку. Запомните: все, что вы поместите в условное выражение оператора `assert`, не вычисляется, когда утверждения выключены. Вы можете создать вспомогательные методы для ваших утверждений, содержащие произвольный объем кода. И хотя это предполагает опасный стиль программирования, вы даже можете использовать утверждения, имеющие побочные эффекты, чтобы отлавливать значения для использования дальнейшими утверждениями — все они будут выключены, когда выключаются утверждения. Например:

```
int savedValue;
assert ( savedValue = getValue()) != -1;
// Произвести работу...
assert checkValue( savedValue );
```

Здесь, в первом операторе `assert`, мы используем вспомогательный метод `getValue()`, чтобы получить информацию и сохранить ее для дальнейшего применения. Затем после выполнения некоторой работы, мы проверяем сохраненное значение, используя другое утверждение и, возможно, сравнивая результаты. Когда утверждения отключены, мы больше не сохраняем и не проверяем данные. Заметьте, что нам необходимо проявить немного хитрости и сделать наше первое условие `assert` булевым, проверяя известное значение. Опять же, использование утверждения с побочными эффектами немного опасно, потому что вам необходимо следить за тем, чтобы эти побочные эффекты были видны только нашим утверждениям. В противном случае вам придется менять поведение вашего приложения, когда вы их отключите.

Массивы

Массив является особым типом объекта, который может хранить упорядоченную коллекцию элементов. Тип элементов массива называется *базовым типом* массива; количество элементов, которое он хранит является фиксированным атрибутом, называемым его *длиной*. Java поддерживает массивы простых и ссылочных типов.

Базовый синтаксис массивов похож на синтаксис в языках C или C++. Мы создаем массив указанной длины и получаем доступ к элементам при помощи *индексного* оператора `[]`. Однако в отличие от других языков, в Java массивы являются истинными объектами первого клас-

са. Массив является экземпляром специального класса Java — `array` – и имеет соответствующий тип в системе типов. Это означает, что, чтобы использовать массив, как и с любыми другими объектами, мы первым делом объявляем переменную подходящего типа, а затем используем оператор `new`, чтобы создать его экземпляр.

Объекты массива отличаются от других объектов в Java в трех отношениях.

- Java косвенно создает специальный тип класса массива для нас, когда мы объявляем новый тип массива. Не обязательно знать об этом процессе, чтобы использовать массивы, но это помогает в дальнейшем понимать их структуру и их отношения с другими объектами в Java.
- Java позволяет нам использовать оператор `[]`, чтобы получить доступы к массиву элементов, так что массивы выглядят, как мы того ожидаем. Мы можем применить наши собственные классы, которые действуют как массивы, но нам пришлось бы согласиться с имением методов `get()` и `set()` вместо использования специального обозначения `[]`.
- Java предоставляет соответствующую специальную форму оператора `new`, который позволяет нам конструировать экземпляр массива указанной длины с обозначением `[]` или инициализировать его непосредственно из структурированного списка значений.

Типы массивов

Переменная типа массива обозначается базовым типом, следующим за пустыми скобками, `[]`. Также Java признает объявление в стиле языка C со скобками после имени массива.

Следующее является эквивалентным:

```
int [] arrayOfInts; // предпочитаемо
int arrayOfInts []; // в стиле C
```

В каждом случае `arrayOfInts` объявляется как массив целых чисел. Размер массива сейчас не является проблемой, поскольку мы объявляем только переменную типа массива. Мы еще не создали фактический экземпляр класса `array` со связанной памятью. Указать длину массива при объявлении переменной типа массива даже невозможно. Размер является строго функцией самого объекта массива, а не ссылкой на него.

Массив ссылочного типа может быть создан таким же образом:

```
String [] someStrings;
Button [] someButtons;
```

Создание и инициализация массива

Оператор `new` используется для создания экземпляра массива. После оператора `new` мы указываем базовый тип массива и его длину при помощи целого выражения в скобках:

```
arrayOfInts = new int [42];
someStrings = new String [ number + 2 ];
```

Мы, конечно, можем комбинировать шаги объявления и назначения массива:

```
double [] someNumbers = new double [20];
Component [] widgets = new Component [12];
```

Индексы массива начинаются с ноля. Таким образом, первый элемент `someNumbers[]` — 0, а последний — 19. После создания элементы массива инициализируются с дефолтными значениями их типа. Для числовых типов это означает, что элементы изначально равны нолю:

```
int [] grades = new int [30];
grades[0] = 99;
grades[1] = 72;
// grades[2] == 0
```

Элементы массива объектов являются ссылками на объекты — как и индивидуальные переменные, на которые они указывают, — но в действительности не содержат экземпляры объектов. Следовательно, дефолтное значение каждого элемента — `null`, пока мы не назначим экземпляры подходящих объектов:

```
String names [] = new String [4];
names [0] = new String();
names [1] = "Boofa";
names [2] = someObject.toString();
// names[3] == null
```

Это важное отличие, которое может вызвать замешательство. Во многих других языках акт создания массива — это то же самое, что и на-

значение хранилища для его элементов. В Java только что назначенный массив объектов в действительности содержит только ссылочные переменные, каждая со значением `null`*. Это не говоря о том, что с пустым массивом память не связана; память необходима для хранения этих ссылок (пустые «слоты» в массиве). Рис. 4.4 иллюстрирует массив `names` из предыдущего примера.

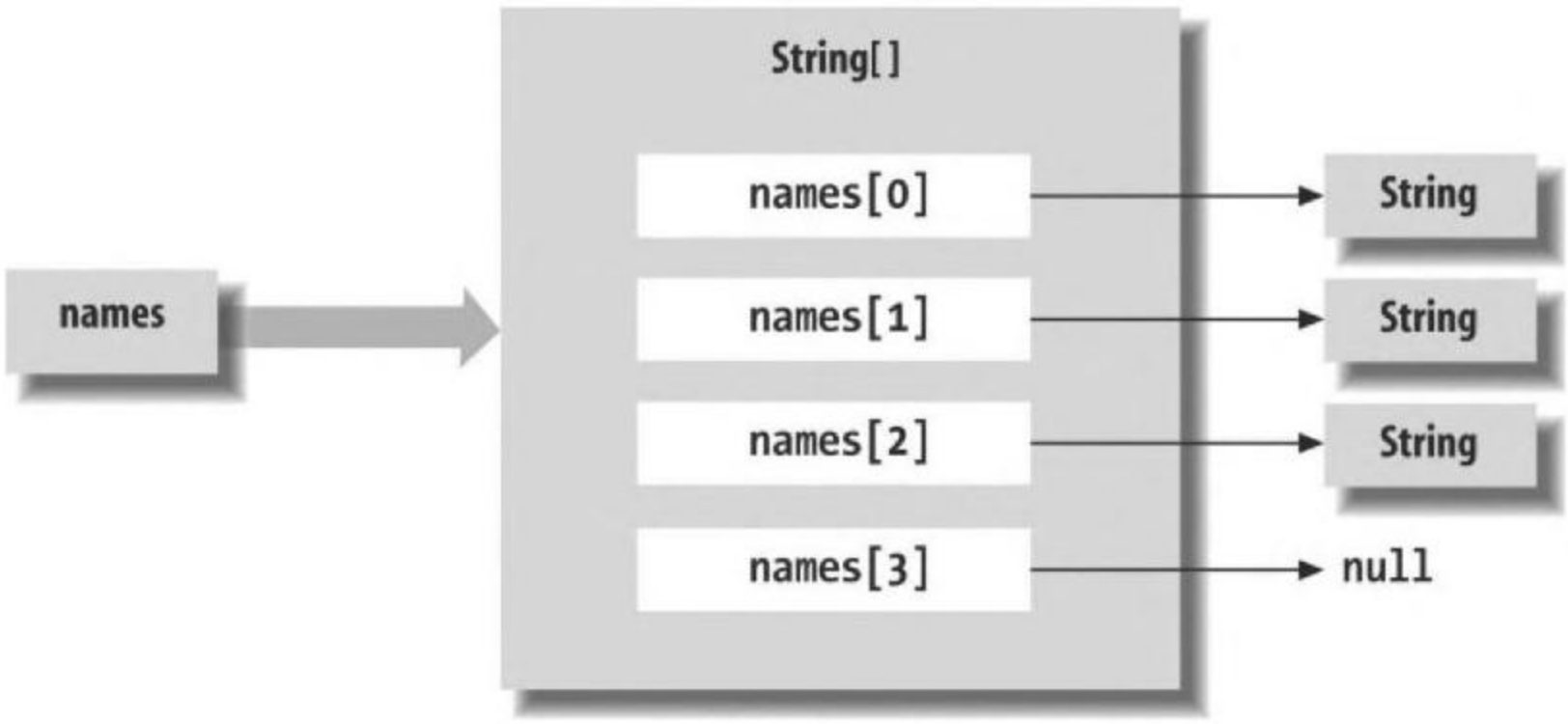

Рис. 4.4. Массив Java

`Names` является переменной типа `String[]` (то есть строчный массив). Этот отдельный объект `String[]` содержит четыре переменных типа `String`. Мы назначили объекты `String` первым трем элементам массива. Четвертый имеет дефолтное значение `null`.

Java поддерживает фигурные скобки в стиле языка C `{}` для создания массива и инициализации его элементов:

```
int [] primes = { 2, 3, 5, 7, 7+4 }; // e.g., primes[2] = 5
```

Объект массива нужного типа и длина создаются косвенно, а значения списка выражений, разделенного запятыми, назначены его элементам. Заметьте, что мы не использовали ключевое слово `new` или тип массива здесь. Тип массива был получен от назначения.

Мы можем использовать синтаксис `{}` с массивом объектов.

---

* Аналог в C или C++ — массив указателей на объекты. Однако указатели в C или C++ сами являются двух- или четырехбитными значениями. Назначение массива указателей является в действительности назначением хранилища для некоторых чисел этих объектов указателей. Массив ссылок концептуально сходен, хотя сами ссылки не являются объектами. Мы не можем управлять ссылками или частями ссылок, кроме как назначением, а требования к их хранению (или нехватка `thereof`) не являются частью спецификации языка Java высокого уровня.

В этом случае каждое выражение должно определять объект, который может быть назначен переменной базового типа массива или значению null.

Вот некоторые примеры:

```
String [] verbs = { "бегать", "прыгать", someWord.toString() };
Button [] controls = { stopButton, new Button("Вперед"),
new Button("Назад") };
// Все типы подтипа Object
Object [] objects = { stopButton, "Слово", null };
```

Следующий код эквивалентен:

```
Button [] threeButtons = new Button [3];
Button [] threeButtons = { null, null, null };
```

Использование массивов

Размер объекта массив доступен в публичной переменной length:

```
char [] alphabet = new char [26];
int alphaLen = alphabet.length; // alphaLen == 26
String [] musketeers = { "один", "два", "три" };
int num = musketeers.length; // num == 3
```

Поле length является единственным доступным полем массива; это переменная, а не метод. (Не беспокойтесь; компилятор сообщит вам, если вы случайно используете скобки в качестве метода, что все иногда делают.)

Доступ к массиву в Java похож на доступ к массиву в других языках; вы можете получить доступ к элементу, поместив выражение с целым значением между скобок после имени массива. Следующий пример создает массив объектов класса Button, называемый keyPad, а затем заполняет массив объектами Button:

```
Button [] keyPad = new Button [ 10 ];
for ( int i=0; i < keyPad.length; i++ )
keyPad[ i ] = new Button( Integer.toString( i ) );
```

Запомните, что мы можем также использовать расширенный цикл for для итерации по значениям массива. Здесь мы используем его для того, чтоб вывести все значения, которые мы только что назначили:

```
for (Button b : keyPad)
System.out.println(b);
```

Попытка получить доступ к элементу, который находится за пределами массива, выдаст исключение `ArrayIndexOutOfBoundsException`. Это тип исключения `RuntimeException`, поэтому вы можете либо отловить его и обработать своими силами, если вы действительно рассчитываете на это, либо игнорировать его, как мы обсуждали ранее:

```
String [] states = new String [50];
try {
states[0] = "Калифорния";
states[1] = "Орегон";
...
states[50] = "Земля Макдональда"; // Ошибка: массив вышел за пределы
}
catch ( ArrayIndexOutOfBoundsException err ) {
System.out.println( "Обработанная ошибка: " + err.getMessage() );
}
```

Распространенной задачей является копирование ряда элементов из одного массива в другой. Одним путем копирования массивов является использование метода `arraycopy()` класса `System` низкого уровня:

```
System.arraycopy( source, sourceStart, destination, destStart, length );
```

Следующий пример удваивает величину массива `names` из предыдущего примера:

```
String [] tmpVar = new String [ 2 * names.length ];
System.arraycopy( names, 0, tmpVar, 0, names.length );
names = tmpVar;
```

Новый массив, размером в два раза больше, чем `names`, присвоен и назначен временной переменной `tmpVar`. Затем используется метод `arraycopy()` для копирования элементов `names` в новый массив. В конце концов, новый массив назначается `names`. Если нет оставшихся ссылок на объект старого массива после того, как `names` был скопирован, он «собирается в мусор» на следующем проходе.

Более легким способом является использование методов `java.util.ArrayscopyOf()` и `copyOfRange()`:

```
byte [] bar = new byte[] { 1, 2, 3, 4, 5 };
byte [] barCopy = Arrays.copyOf( bar, bar.length );
// { 1, 2, 3, 4, 5 }
byte [] expanded = Arrays.copyOf( bar, bar.length+2 );
```

```
// { 1, 2, 3, 4, 5, 0, 0 }
byte [] firstThree = Arrays.copyOfRange( bar, 0, 3 );
// { 1, 2, 3 }
byte [] lastThree = Arrays.copyOfRange( bar, 2, bar.length );
// { 3, 4, 5 }
byte [] lastThreePlusTwo = Arrays.copyOfRange( bar, 2, bar.length+2 );
// { 3, 4, 5, 0, 0 }
```

Метод `copyOf()` берет исходный массив и целевую длину. Если целевая длина больше, чем длина исходного массива, то новый массив перегружается (нолями и значениями `null`) до необходимой длины. `copyOfRange()` берет начальный индекс (включающий) и конечный индекс (исключающий) и желаемую ширину, которая будет увеличена, если необходимо.

Анонимные массивы

Часто удобно создавать «одноразовые» массивы, которые используются в одном месте и на которые нет ссылок нигде больше. Таким массивам не нужно имя, поскольку вам никогда не понадобится снова ссылаться на них в этом контексте. Например, вам может захотеться создать коллекцию объектов, чтобы передать в качестве аргумента какому-то методу. Довольно легко создать нормальный массив с именем, но если вы действительно больше не будете работать с этим массивом (если вы будете использовать массив только как хранилище для какой-то коллекции), вам нет необходимости это делать. В Java легко создать «анонимные» (безымянные) массивы.

Скажем, вам нужно вызвать метод, называющийся `setPets()`, который принимает массив объектов `Animal` в качестве аргументов. Если `Cat` и `Dog` являются подклассами `Animal`, вот как можно использовать `setPets()`, используя анонимный массив:

```
Dog pokey = new Dog ("серый");
Cat boojum = new Cat ("серый");
Cat simon = new Cat ("рыжий");
setPets ( new Animal [] { pokey, boojum, simon });
```

Синтаксис выглядит так же, как инициализация массива в объявлении переменной. Косвенно мы определяем величину массива и наполняем его элементами, используя обозначение в фигурных скобках. Однако, поскольку это не объявление переменной, мы должны четко использовать оператор `new` и тип массив для создания объекта массив.

Анонимные массивы иногда использовались как замена списка аргументов шириной в переменную к методам, которые обсуждаются в главе 5. С введением списка аргументов длины переменной в Java использование анонимных массивов сократилось.

Многомерные массивы

Java поддерживает многомерные массивы в форме массивов объектов типа массив. Вы создаете многомерный массив с синтаксисом подобным C, используя множество пар скобок, одна для каждого измерения. Вы также используете этот синтаксис для доступа к элементам в разных положениях внутри массива. Вот пример многомерного массива, который представляет шахматную доску:

```
ChessPiece [][] chessBoard;
chessBoard = new ChessPiece [8][8];
chessBoard[0][0] = new ChessPiece.Rook;
chessBoard[1][0] = new ChessPiece.Pawn;
...
```

Здесь `chessBoard` объявлен как переменная типа `ChessPiece[][]` (то есть массив массивов `ChessPiece`). Это объявление косвенно создает также тип `ChessPiece[]`. Пример также иллюстрирует специальную форму оператора `new`, использующегося для создания многомерного массива. Он создает массив объектов `ChessPiece[]`, а затем в свою очередь делает каждый элемент массивом объектов `ChessPiece`. Затем мы индексируем `chessBoard`, чтобы определить значения для отдельных элементов `ChessPiece`. (Здесь мы не будем обращать внимание на цвет кусков кода.)

Конечно, вы можете создать массивы с более чем двумя измерениями. Вот немного непрактичный пример:

```
Color [][][] rgbCube = new Color [256][256][256];
rgbCube[0][0][0] = Color.black;
rgbCube[255][255][0] = Color.yellow;
...
```

Мы можем указать частичный индекс многомерного массива, чтобы получить подмассив объектов типа массив с меньшим количеством измерений. В нашем примере переменная `chessBoard` имеет тип `ChessPiece[][]`.Выражение `chessBoard[0]` является правильным и ссылается на первый

элемент chessBoard, который в Java имеет тип ChessPiece[]. Например, мы можем заполнить один ряд нашей шахматной доски за один раз:

```
ChessPiece [] homeRow = {
new ChessPiece("Ладья"), new ChessPiece("Конь"),
new ChessPiece("Слон"), new ChessPiece("Король"),
new ChessPiece("Ферзь"), new ChessPiece("Слон"),
new ChessPiece("Конь"), new ChessPiece("Ладья")
};
chessBoard[0] = homeRow;
```

У нас нет необходимости указывать размер измерений многомерного массива одним оператором new. Синтаксис оператора new позволяет нам оставить величину некоторых измерений неопределенной. Размер хотя бы первого измерения (самого важного измерения в массиве) должен быть указан, но размеры любого количества следующих, менее важных измерений массива могут остаться неопределенными. Мы можем назначить подходящие значения массивного типа позже.

Мы можем создать шахматную доску булевых значений (чего будет недостаточно для настоящей игры в шахматы), используя эту технику:

```
boolean [][] checkerBoard;
checkerBoard = new boolean [8][];
```

Здесь массив checkerBoard объявлен и создан, но его элементы, восемь объектов boolean[] следующего уровня, оставлены пустыми. Так, например, checkerBoard[0] имеет значение null, пока мы ясно не создадим массив и назначим его как указано далее:

```
checkerBoard[0] = new boolean [8];
checkerBoard[1] = new boolean [8];
...
checkerBoard[7] = new boolean [8];
```

Код предыдущих двух примеров эквивалентен этому:

```
boolean [][] checkerBoard = new boolean [8][8];
```

Одна из причин, по которой мы можем захотеть оставить измерения массива неуказанными, — это то, что мы сможем хранить массивы, полученные нами от другого метода.

Отметьте это, поскольку длина массива не является частью этого типа, массивы в шахматной доске не обязательно должны иметь оди-

наковую длину; то есть многомерные массивы не обязаны быть прямоугольными. Это неполная (но полностью законная в Java) шахматная доска:

```
checkerBoard[2] = new boolean [3];
checkerBoard[3] = new boolean [10];
```

А вот как вы могли бы создать и назначить треугольный массив:

```
int [][] triangle = new int [5][];
for (int i = 0; i < triangle.length; i++) {
triangle[i] = new int [i + 1];
for (int j = 0; j < i + 1; j++)
triangle[i][j] = i + j;
}
```

Внутри массивов

Ранее мы сказали, что массивы являются экземплярами специальных классов массивов в языке Java. Если у массивов в Java есть классы, где они должны находиться в иерархии классов и как они связаны? Это хорошие вопросы, но перед тем как ответить на них, мы должны еще поговорить об объектно-ориентированных аспектах Java. Это тема следующих глав. Сейчас примите на веру то, что массивы встраиваются в иерархию классов.

Материал, защищенный авторским правом

Глава 5
ОБЪЕКТЫ В JAVA

В этой главе мы доберемся до сердца Java и исследуем объектно-ориентированные аспекты языка. Понятие *объектно-ориентированное проектирование* относится к искусству разбора приложения на некоторое количество объектов, являющихся автономными компонентами приложения, функционирующими в единой связке. Целью является разбить вашу проблему на более мелкие проблемы, которые проще и легче обрабатывать и которыми легче управлять. Объектно-ориентированное проектирование оправдало себя за многие годы, и объектно-ориентированные языки, такие как Java, предоставляют мощный фундамент для написания приложений — от очень малых до очень больших. Java во всех отношениях был разработан как объектно-ориентированный язык, и в основе всех его API-интерфейсов и библиотек лежит прочный объектно-ориентированный фундамент.

«Методология» объектного проектирования является системой или набором правил, которые помогут разбить ваше приложение на объекты. Часто это означает отображение субъектов и идей (иногда называемых «проблемными областями») реального мира в компонентах приложения. Различные методологии часто стараются помочь вам разложить приложение на хороший набор многократно используемых объектов. В принципе это хорошо, но проблема заключается в том, что хорошее объектно-ориентированное проектирование до сих пор является больше искусством, чем наукой. Вы можете изучать разнообразные стандартные методологии проектирования, но ни одна из них не поможет вам во всех ситуациях. Правда заключается в том, что ничто не заменит опыта.

Мы не будем стараться подтолкнуть вас к какой-то определенной методологии; для этого* существуют целые полки с книгами. Вместо

---

* Когда у вас будет некоторый опыт работы с базовыми объектно-ориентированными концепциями, возможно, вам захочется заглянуть в книгу «Приемы объектно-ориентированного проектирования. Паттерны проектирования» // Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес («Питер», 2007). В этой книге собраны полезные объектно-ориентированные проекты, которые были улучшены за годы опыта. Многие появляются в проектировании API Java.

этого мы предлагаем некоторые продиктованные здравым смыслом подсказки, чтобы помочь вам начать. Надеемся, что следующие общие руководства по проектированию покажутся вам более осмысленными после того, как вы прочтете эту и следующую главы.

- Прячьте как можно больше вашей реализации. Никогда не показывайте больше свойств объекта, чем вам нужно. Это ключ к созданию управляемого, разумного кода. Избегайте публичных переменных в ваших объектах с возможностью исключения только для констант. Вместо этого определяйте методы *доступа*, чтобы устанавливать и возвращать значения (даже если они являются простыми типами). Позже, когда вам понадобится, вы сможете изменять и расширять поведение ваших объектов, не нарушая другие классы, которые полагаются на них.
- Создавайте новые объекты, только когда это необходимо — используйте *соединение* вместо *наследования*. Когда вы используете объект в его существующей форме как часть нового объекта, вы *создаете* объекты. Когда вы изменяете или улучшаете поведение объекта (при помощи *дочерних* классов), вы используете *наследование*. Вы должны пытаться повторно использовать объекты соединением, а не наследованием, когда это возможно, поскольку, когда вы соединяете объекты, вы получаете полную выгоду от существующих инструментов. Наследование влечет за собой нарушение инкапсуляции объекта и должно производиться только тогда, когда от этого можно получить реальную выгоду. Спросите себя, действительно ли вам нужно наследовать весь публичный интерфейс объекта (хотите ли вы иметь что-то «похожее» на этот объект?) или вы хотите только делегировать определенную работу объекту и использовать его путем соединения.
- Минимизируйте отношения между объектами и старайтесь организовать объекты в пакеты. Классы, которые работают вплотную друг с другом, можно сгруппировать, используя пакеты Java, и таким образом спрятать то, что не представляет общего интереса. Выставляйте классы, только если хотите, чтобы их использовали другие люди. Чем слабее связаны ваши объекты, тем легче использовать их снова впоследствии.

Классы

Классы являются строительными блоками приложения Java. *Класс* может содержать методы (функции), переменные, код инициализации и, как мы увидим позже, другие классы. Он служит шаблоном для соз-

дания *экземпляров* класса, которые являются объектами (индивидуальными копиями), реализующими структуру класса. Вы объявляете класс при помощи ключевого слова `class`. Методы и переменные класса указываются в скобках объявления класса:

```
class Pendulum {
float mass;
float length = 1.0f;
int cycles;
float getPosition ( float time ) {
...
}
...
}
```

Класс `Pendulum` содержит три переменных: `mass`, `length` и `cycles`. Он также определяет метод, называемый `getPosition()`, который принимает значение типа `float` в качестве аргумента и возвращает значение `float` в качестве результата. Переменные и объявление методов могут быть указаны в любом порядке, но инициализаторы переменных не могут создать «прямых ссылок» на другие переменные, которые появляются позже. Как только мы определили класс `Pendulum`, мы можем создать объект `Pendulum` (экземпляр класса) следующим образом:

```
Pendulum p;
p = new Pendulum();
```

Повторный вызов нашего объявления переменной p не создает объект `Pendulum`, он просто создает переменную, которая ссылается на объект типа `Pendulum`. Нам все же пришлось создать объект, используя ключевое слово `new`, как показано во второй строке предыдущего участка кода. Теперь, когда мы создали объект `Pendulum`, мы можем получить доступ к его переменным и методам, как мы видели много раз:

```
p.mass = 5.0;
float pos = p.getPosition( 1.0 );
```

Два типа переменных могут быть определены в классе: *переменная экземпляра* и *статическая переменная*. Каждый экземпляр объекта имеет свой набор переменных экземпляра; значения этих переменных в одном экземпляре объекта могут отличаться от значений в другом объекте. Позже мы поговорим о статических переменных, которые, для сравнения, являются общими для всех экземпляров объекта. В любом

случае, если вы не инициализируете переменную, когда объявите ее, ей присваивается значение по умолчанию, подходящее для ее типа (`null`, `ноль` или `false`).

Рис. 5.1 демонстрирует гипотетическое приложение `TextBook`, которое использует два экземпляра класса `Pendulum` через ссылочные переменные `bigPendulum` и `smallPendulum`. Каждый из этих объектов `Pendulum` имеет свою копию `mass`, `length` и `cycles`. Как и переменные, методы, определенные в классе, могут быть *методами экземпляра* или *статическими методами*. Метод экземпляра связан только с одним экземпляром класса, но отношение не такое простое, как для переменных. Методы экземпляра доступны через экземпляр объекта, но объект на самом деле не имеет собственной «копии» методов (нет дублирования кода). Вместо этого связанность означает, что методы экземпляра могут «видеть» значения переменных экземпляра объекта и управлять ими.

В главе 6, когда мы будем говорить о подклассах, вас ждет еще много информации о том, как методы видят переменные. В той главе мы также обсудим, как методы экземпляра «перезаписываются» в дочерних классах — очень важное свойство объектно-ориентированного проектирования. Оба аспекта отличаются от статических методов, которые, как мы увидим, скорее похожи на глобальные функции, поскольку они ассоциируются только с именем класса.

Доступ к полям и методам

Внутри класса мы можем получить доступ к переменным и методам вызова класса непосредственно по имени. Вот пример, который расширяет наш класс `Pendulum`:

```
class Pendulum {
...
void resetEverything() {
mass = 1.0;
length = 1.0;
cycles = 0;
...
float startingPosition = getPosition( 0.0 );
}
...
}
```

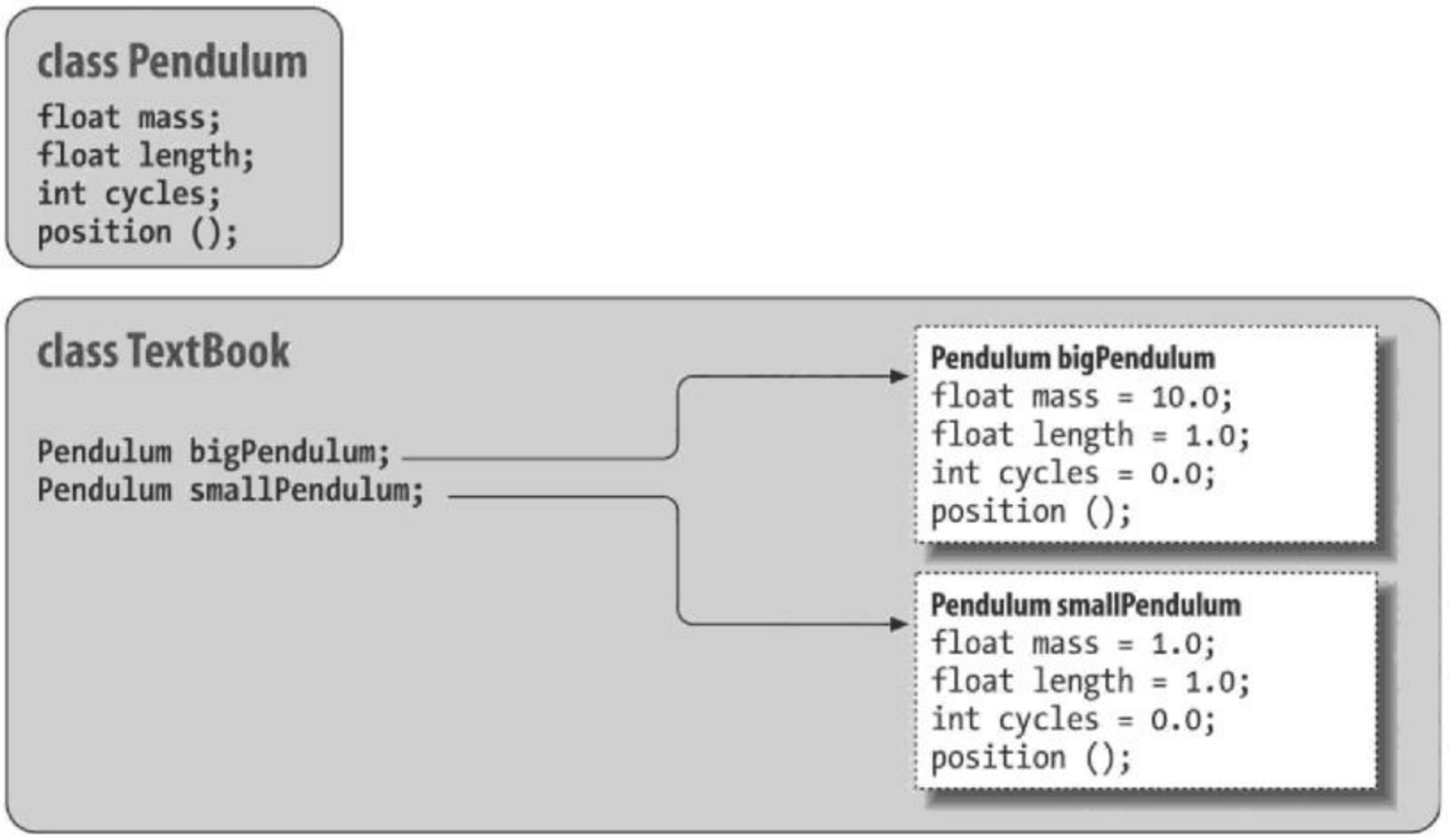

Рис. 5.1. Экземпляры класса `Pendulum`

Другие классы получают доступ к объекту через ссылку, используя обозначение селектора-точки, которое мы обсуждали в предыдущей главе:

```
class TextBook {
...
void showPendulum() {
Pendulum bob = new Pendulum();
...
int i = bob.cycles;
bob.resetEverything();
bob.mass = 1.01;
...
}
...
}
```

Здесь мы создали второй класс, `TextBook`, который использует объект `Pendulum`. Он создает экземпляр в `showPendulum()`, а затем вызывает методы и получает доступ к переменным объекта через ссылку `bob`.

Несколько факторов влияет на то, будут ли члены класса доступны из другого класса. Вы можете использовать модификаторы видимости `public`, `private` и `protected` для контроля доступа; классы также могут быть помещены в *пакет*, который влияет на их область видимости. Модификатор `public`, например, назначает переменную или метод для

использования только другими членами самого класса. В предыдущем примере мы могли бы сделать нашу переменную `cycles` частной, используя модификатор `private`:

```
class Pendulum {
...
private int cycles;
...
```

Теперь мы не можем получить доступ к `cycles` из `TextBook`:

```
class TextBook {
...
void showPendulum() {
...
int i = bob.cycles; // ошибка во время компиляции
```

Если нам все еще нужно получить доступ к `cycles` в каком-то качестве, мы можем добавить публичный метод `getCycles()` к классу `Pendulum`. (Создание методов доступа, подобных этому, является хорошим правилом проектирования, поскольку оно позволяет дальнейшую гибкость в смене типа поведения значения.) Мы подробно рассмотрим пакеты, модификаторы доступа и то, как они влияют на видимость переменных и методов в главе 6.

Статические члены

Как мы сказали, переменные и методы экземпляра связаны с экземпляром класса (то есть через отдельный объект, как `bob` в предыдущем примере) и доступны через него. Для сравнения: члены, которые объявляются через модификатор `static`, находятся в классе и делятся всеми экземплярами класса. Переменные, объявляемые модификатором `static`, называются *статическими переменными*, или *переменными класса*; аналогично, эти типы методов называются *статическими методами* или *методами класса*. Мы можем добавить статическую переменную в наш пример `Pendulum`:

```
class Pendulum {
...
static float gravAccel = 9.80;
...
```

Мы объявили новую переменную `gravAccel` класса `float` как `static`. Это означает, что она связана с классом, а не с отдельным экземпляром и если мы изменим ее значение (либо прямо, либо через любой экзем-

пляр Pendulum), значение изменится для всех объектов Pendulum, как показано на рис. 5.2.

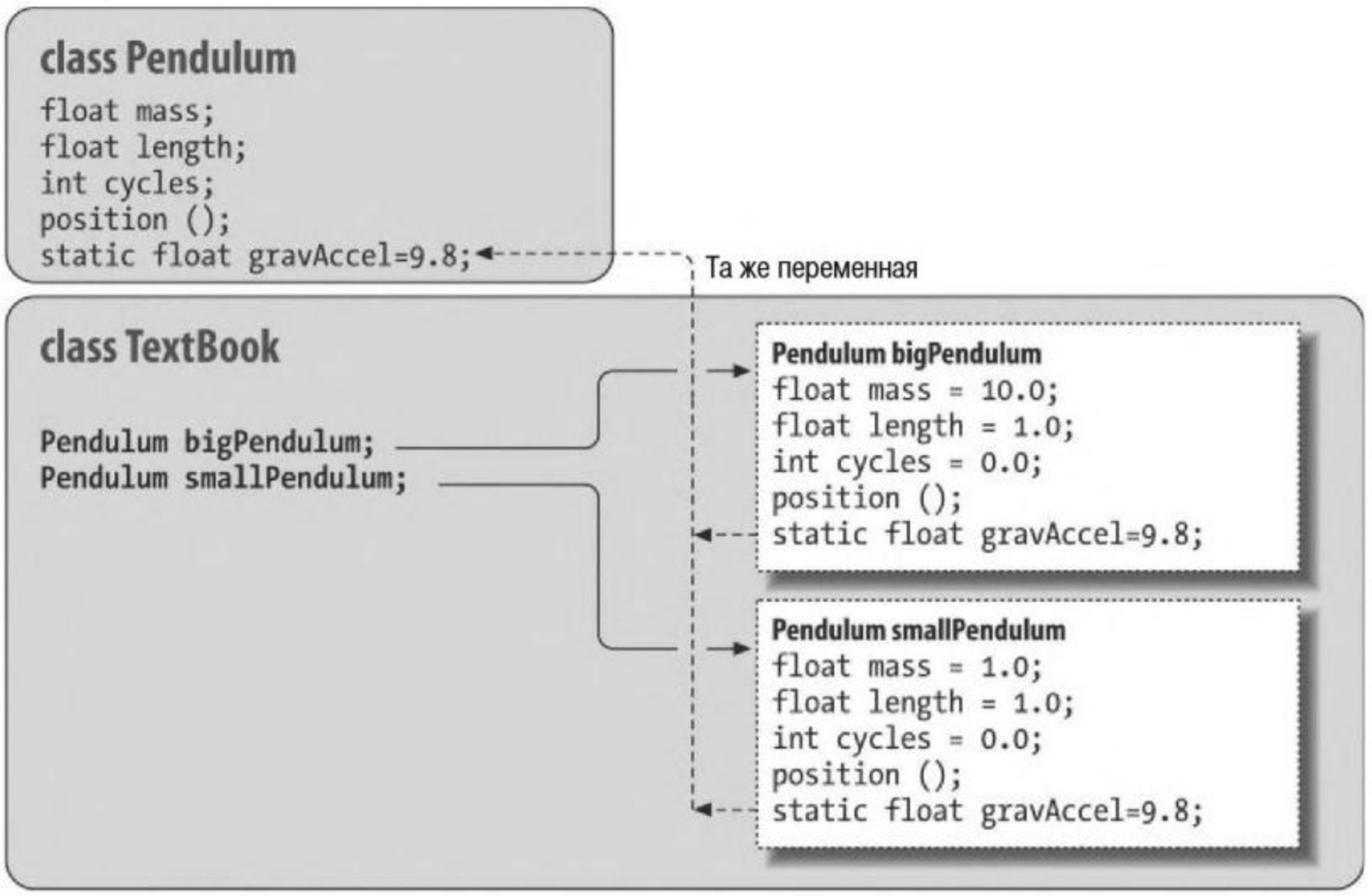

Рис. 5.2. Статические переменные, делимые всеми экземплярами класса

Статические переменные могут быть доступны как члены экземпляра. Внутри нашего класса Pendulum мы можем сослаться на переменную gravAccel, как на любую другую переменную:

```
class Pendulum {
...
float getWeight () {
return mass * gravAccel;
}
...
}
```

Однако, поскольку статические члены существуют в самом классе, независимо от экземпляра, мы также можем получить доступ к ним непосредственно через класс. Нам не нужен объект Pendulum, чтобы получить или установить переменную gravAccel; вместо этого мы можем использовать класс, чтобы выбрать переменную:

```
Pendulum.gravAccel = 8.76;
```

Это меняет значение переменной `gravAccel` на видимое для всех экземпляров. Зачем нам нужно изменять значение `gravAccel`? Что ж, возможно, нам захочется узнать, как работают маятники на разных планетах. Статические переменные также очень полезны для других типов данных, которые делятся среди классов во время выполнения. Например, вы можете создать методы для регистрации ваших экземпляров объекта так, что они смогут взаимодействовать, или так, чтобы вы могли следить за ними всеми. Также распространено использование статических переменных для определения постоянных значений. В этом случае мы используем модификатор `static` вместе с модификатором `final`. Итак, если бы нас беспокоило только то, как ведут себя маятники под действием гравитации Земли, мы бы могли изменить класс `Pendulum` следующим образом:

```
class Pendulum {
...
static final float EARTH_G = 9.80;
...
```

Здесь мы последовали общепринятому обычаю и назвали нашу константу с прописной буквы. Значение `EARTH_G` является константой; оно может быть доступно через класс `Pendulum` или его экземпляры, но его нельзя изменить во время выполнения.

Важно использовать комбинацию методов `static` и `final` для вещей, которые действительно являются константами. Это потому, что компилятору позволено «подключать» такие значения в классах, которые ссылаются на них. Это означает, что если вы измените переменную `static final`, вам придется перекомпилировать весь код, который использует этот класс (это действительно единственный случай, когда вам приходится так поступать в Java). Статические члены полезны как флаги и идентификаторы, к которым можно получить доступ отовсюду. Они также полезны в значениях, которые необходимы в конструкции самих экземпляров. В нашем примере мы можем объявить целый ряд статических значений, чтобы представить разнообразные типы объектов `Pendulum`:

```
class Pendulum {
...
static int SIMPLE = 0, ONE_SPRING = 1, TWO_SPRING = 2;
...
```

Затем мы можем использовать эти флаги в методе, который устанавливает тип `Pendulum`, или в специальном конструкторе, как мы обсудим вскоре:

```
Pendulum pendy = new Pendulum();
pendy.setType( Pendulum.ONE_SPRING );
```

Опять же, внутри класса `Pendulum` мы можем использовать статические члены непосредственно по имени; нет необходимости в префиксе `Pendulum.`:

```
class Pendulum {
...
void resetEverything() {
setType ( SIMPLE );
...
}
```

Выбор между константами и перечислениями

В предыдущем разделе мы видели два примера использования статически финальных переменных (констант). Первым было создание настоящих констант; в том случае это была числовая константа `EARTH_G`, но она также с легкостью могла бы быть значением класса `String` или `Date`. Вторым применением было создание фиксированного набора идентификаторов `SIMPLE`, `ONE_SPRING` и др., чьи настоящие значения не были так важны, как их уникальность и, возможно, их определенный порядок.

Перечисления были добавлены в язык Java для замены этого идентификатора механизмом, который и безопаснее, и в некоторых случаях более эффективен. Мы могли бы объявить типы нашего маятника таким образом:

```
public enum PendulumTypes { Simple, OneSpring, TwoSpring }
```

Это перечисление создает не только значения, но и новый тип, `PendulumTypes`, чье значение ограничено до одного из трех дискретных идентификаторов. Вызывающий код может ссылаться на значения, как он делал через наш класс: `PendulumTypes.Simple`. Мы изменили регистр в коде, чтобы отойти от соглашения для целочисленных констант, но вы можете придерживаться верхнего регистра, если вам так нравится.

Далее, когда мы будем говорить об импорте классов и пакетов, мы обсудим свойство *статического импорта* в Java, позволяющее нам импортировать статические идентификаторы и перечисления (которые,

как мы видим, связаны) в класс, так что мы можем использовать их с простыми именами. Например:

```
new Pendulum(OneSpring );
```

Мы перейдем к деталям перечислений позже в этой главе, когда рассмотрим объекты более подробно.

Методы

Методы размещаются внутри тела класса. Они содержат объявление локальных переменных и другие операторы Java, которые выполняются, когда вызывается метод. Методы могут вернуть значение вызывающему оператору. Они всегда уточняют тип возврата, который может быть простым типом, ссылочным типом или типом `void`, который означает, что нет возвращаемого значения. Методы могут принимать аргументы, которые являются значениями, поставляемыми вызываемым оператором метода.

Вот постой пример:

```
class Bird {
int xPos, yPos;
double fly ( int x, int y ) {
double distance = Math.sqrt( x*x + y*y );
flap( distance );
xPos = x;
yPos = y;
return distance;
}
...
}
```

В этом примере класс `Bird` определяет метод `fly()`, который берет в качестве аргументов два целых числа: `x` и `y`. Он возвращает значение типа `double` в качестве результата, используя ключевое слово `return`.

У нашего метода фиксированное количество аргументов (два); однако методы могут иметь *списки аргументов длиной с переменную*, и это позволяет методу уточнять, что он может брать любое количество аргументов и самостоятельно сортировать их во время выполнения. Далее в этой главе мы рассмотрим это детальнее.

Локальные переменные

Наш метод `fly()` объявляет локальную переменную под названием `distance`, которую он использует для вычисления пройденной дистанции. Локальная переменная является временной; она существует только в рамках (в блоке) этого метода. Локальные переменные создаются, когда вызывается метод; они обычно разрушаются, когда метод возвращается. На них невозможно ссылаться вне самого метода. Если метод выполняется параллельно в разных потоках, каждый поток имеет свою версию локальных переменных метода. Аргументы метода также служат локальными переменными в рамках метода; единственной разницей является то, что они инициализируются при передаче от вызывающего оператора метода.

Объект, созданный внутри метода и назначенный локальной переменной, может продолжать или не продолжать существовать после возвращения метода. Как и со всеми объектами Java, это зависит от того, остались ли ссылки на объект. Если объект создан, назначен локальной переменной и нигде никогда не используется, этот объект больше не является ссылаемым, когда локальная переменная исчезает из границ, поэтому чистка памяти убирает объект. Однако, если мы назначаем объект переменной экземпляра объекта, передаем ее в качестве аргумента другому методу или передаем ее назад в качестве возвратной переменной, она может сохраниться другой переменной, содержащей его ссылку. Вскоре мы рассмотрим создание объекта и чистку памяти более подробно.

Сокрытие

Если локальная переменная и переменная экземпляра имеют одно и то же имя, локальная переменная *скрывает* имя переменной экземпляра в рамках границ метода. В следующем примере локальные переменные `xPos` и `yPos` прячут переменные экземпляра с тем же именем:

```
class Bird {
int xPos, yPos;
int xNest, yNest;
...
double flyToNest() {
int xPos = xNest;
int yPos = yNest:
```

```
return ( fly( xPos, yPos ) );
}
...
}
```

Когда мы устанавливаем переменные локальных переменных в `flyToNest()`, это не влияет на значение переменных экземпляра.

Ссылка this

Вы можете использовать специальную ссылку `this` в любой момент, когда вам нужно сослаться непосредственно на текущий объект или член текущего объекта. Часто вам не нужно использовать `this`, поскольку ссылка на текущий объект косвенная; таким является случай с использованием однозначно названных переменных экземпляра внутри класса. Но мы можем использовать `this`, чтобы ясно сослаться на переменные экземпляра в нашем объекте, даже если они скрыты. Следующий пример показывает, как мы можем использовать `this`, чтобы разрешить имена аргументов, которые скрывают имена переменных экземпляра. Это достаточно распространенная техника, поскольку она спасает от необходимости придумывать альтернативные имена. Вот как мы можем применить наш метод `fly()` с сокрытыми переменными:

```
class Bird {
int xPos, yPos;
double fly ( int xPos, int yPos ) {
double distance = Math.sqrt( xPos*xPos + yPos*yPos );
flap( distance );
this.xPos = xPos; // экземпляр var = локальная переменная vra
this.yPos = yPos;
return distance;
}
...
}
```

В этом примере выражение this.xPos ссылается на переменную экземпляра xPos и присваивает ей значение локальной переменной xPos, которая в противном случае скроет его имя. Единственной причиной, по которой мы используем this в предыдущем примере, является то, что мы используем имена аргумента, скрывающие наши переменные экземпляра, и мы хотим сослаться на переменные экземпляра. Вы также можете использовать ссылку this всегда, когда захотите передать ссылку

на «текущий» ограничивающий объект какому-то другому методу; мы покажем вам примеры этого.

Статические методы

Статические методы (методы класса), как и статические переменные, принадлежат классу, а не отдельным экземплярам класса. Что это значит? Что ж, прежде всего статический метод существует вне какого-либо отдельного экземпляра класса. Он может вызываться по имени, через имя класса, без каких-либо объектов вокруг. Поскольку он не граничит с отдельными экземплярами объекта, статический метод может непосредственно получить доступ только к другим статическим методам (статическим переменным или другим статическим методам) класса. Он не может непосредственно видеть переменные экземпляра или вызывать методы экземпляра, поскольку, чтобы сделать это, нам нужно спросить: «к какому экземпляру?» Статические методы могут вызываться из экземпляров, синтаксически как методы экземпляров, но важно то, что они также могут использоваться независимо.

Наш метод `fly()` использует статический метод: `Math.sqrt()`, который определяется классом `java.lang.Math`; мы исследуем этот класс в подробностях в главе 11. Сейчас вам нужно запомнить важную вещь: `Math` — это имя класса, а не экземпляр объекта `Math`. (Так уж получилось, что вы даже не можете создать экземпляр класса `Math`.) Поскольку статические методы могут вызываться всегда, когда доступно имя класса, методы класса больше похожи на функции в стиле языка C. Статические методы особо полезны для методов утилит, которые производят работу, полезную либо независимо от экземпляров, либо при работе с экземплярами. Например, в нашем классе `Bird` мы можем перечислять все доступные типы птиц, которые могут быть созданы:

```
class Bird {
...
static String [] getBirdTypes() { ... }
}
```

Здесь мы определили статический метод `getBirdTypes()`, который возвращает массив строк, содержащих названия птиц. Мы можем использовать `getBirdTypes()` из экземпляра `Bird` как метод экземпляра. Однако мы можем вызывать его из других классов, используя имя класса `Bird`:

```
String [] names = Bird.getBirdTypes();
```

Возможно, специальная версия конструктора класса `Bird` принимает имя типа птицы. Мы можем использовать этот список, чтобы решить, какой тип птицы создать.

Статические методы также играют важную роль в разнообразных шаблонах проектов, где вы ограничиваете использование оператора `new` для класса до одного метода — статического, называемого *фабричным методом*. Мы поговорим больше о конструкторах объекта позже, но будет достаточно сказать, что такое использование является распространенным:

```
Bird bird = Bird.createBird( "голубь" );
```

Инициализация локальных переменных

В примере `flyToNest()` мы посчитали обязательным инициализировать локальные переменные `xPos` и `yPos`. В отличие от переменных экземпляра локальные переменные должны быть инициализированы перед использованием. Попытка получить доступ к локальной переменной , которая не содержит значения, приводит к ошибке на этапе компиляции:

```
void myMethod() {
int foo = 42;
int bar;
bar += 1; // ошибка компиляции, bar не инициализирована
bar = 99;
bar += 1; // все нормально

    }
```

Заметьте: это не предполагает, что локальные переменные должны быть инициализированы при объявлении, только то, что они должны быть назначены, когда на них впервые ссылаются. Менее явные возможности возникают при создании назначений внутри условных операторов:

```
void myMethod {
int foo;
if ( someCondition ) {
foo = 42;
...
}
foo += 1; // Ошибка во время компиляции, возможно, foo не инициализирована
}
```

В этом примере переменная `foo` инициализируется, только если условие `someCondition` истинно. Компилятор не позволяет вам сделать на это ставку, поэтому он помечает использование `foo` как ошибку. Мы можем исправить эту ситуацию несколькими способами. Мы можем инициализировать переменную дефолтным значением заранее или переместить использование внутрь условного оператора. Мы также можем убедиться, что путь выполнения не достигнет неинициализированной переменной другим способом, в зависимости от того, что целесообразно для нашего отдельного приложения. Например, мы могли бы просто убедиться, что назначаем переменной `foo` значение в ветках `if` и `else`. Или мы можем резко вернуться из метода:

```
int foo;
...
if ( someCondition ) {
foo = 42;
...
} else
return;
foo += 1;
```

В этом случае нет вероятности, что мы получим переменную `foo` в неинициализированном состоянии, поэтому компилятор позволяет использование `foo` после условного оператора.

Почему язык Java такой привередливый в отношении локальных переменных? Один из самых распространенных (и коварных) источников ошибок в языке C или C++ — это пренебрежение инициализацией локальных переменных, поэтому Java пытается прийти на помощь. Если бы этого не происходило, Java бы подвергалась тем же потенциальным нарушениям порядка, что и в C или C++.*

Передача аргументов и ссылки

В начале главы 4 мы описывали отличие между простыми типами, которые передаются через значение (при помощи копирования), и объек-

* Как в хранилище `malloc` в C или C++, объекты Java и их переменные экземпляра локализуются в динамической памяти, что позволяет им получать значения по умолчанию сразу при создании. Однако локальные переменные локализуются в стеке виртуальной машины Java. Как в стеке C и C++ ошибка их инициализации может означать , что удачные вызовы методов могут получить значения из «мусора», а выполнение программы может быть противоречивым или зависимым от реализации.

тами, которые передаются при помощи ссылок. Теперь, когда у нас есть лучший контроль над методами в Java, давайте рассмотрим пример:

```
void myMethod( int j, SomeKindOfObject o ) {
...
// использование объекта
int i = 0;
SomeKindOfObject obj = new SomeKindOfObject();
myMethod( i, obj );
```

Этот участок кода вызывает метод `myMethod()`, передает ему два аргумента. Первый аргумент, `i`, передается по значению; когда метод вызывается, значение `i` копируется в параметр метода (локальную переменную к нему), называемую `j`. Если `myMethod()` меняет значение `j`, он меняет только копию локальной переменной.

Таким же образом копия ссылки на `obj` помещается в ссылочную переменную `o` метода `myMethod()`. Обе ссылки ссылаются на один и тот же объект, поэтому любые изменения, производимые в любой из ссылок, влияют на фактический (единственный) экземпляр объекта. Если мы изменим значение, скажем, `o.size`, изменение будет заметно как в `o.size` (внутри `myMethod()`), так и в `obj.size` (в вызывающем объекте). Однако если `myMethod()` изменит саму ссылку `o`, чтобы она указывала на другой объект, это влияет только на ссылку его локальной переменной. Это не влияет на переменную вызывающего оператора `obj`, который все еще ссылается на исходный объект. В этом отношении передача ссылки подобна передаче указателя в языке C и не похожа на передачу по ссылке, как в языке C++.

Что если `myMethod()` нуждается в изменении обозначения ссылки `obj` в вызывающем методе (то есть сделать, чтобы `obj` указывал на другой объект)? Легкий способ сделать это — поместить `obj` внутрь объекта какого-то типа. Например, мы можем сделать из объекта единственный элемент массива:

```
SomeKindOfObject [] wrapper = new SomeKindOfObject [] { obj
};
```

Все стороны смогут затем сослаться на объект `wrapper[0]` и будут иметь способность изменять ссылку. Это эстетически неприятно, но демонстрирует, что нам необходим промежуточный уровень.

Другая возможность — это использование `this` для передачи ссылки вызывающему объекту. В этом случае вызывающий объект служит

оберткой для ссылки. Давайте посмотрим на кусок кода, который может быть частью реализации списка указателей:

```
class Element {
public Element nextElement;
void addToList( List list ) {
list.addToList( this );
}
}
class List {
void addToList( Element element ) {
...
element.nextElement = getNextElement();
}
}
```

Каждый элемент в списке указателей содержит указатель на следующий элемент в списке. В этом коде класс `Element` представляет один элемент; он включает метод для добавления его самого в список. Сам класс `List` содержит метод для добавления произвольного класса `Element` в список. Метод `addToList()` вызывает `addToList()` с аргументом `this` (который, конечно, является классом `Element`). `addToList()` может использовать ссылку `this`, чтобы изменить переменную экземпляра `nextElement` класса `Element`. Та же самая техника может использоваться совместно с интерфейсами для реализации обратной связи для вызова случайного метода.

Обертки для простых типов

Как мы описывали в главе 4, в мире Java существует разделение между типами классов (то есть объектами) и простыми типами (то есть числами, символами, булевыми значениями). Java соглашается на компромисс просто ради эффективности. Для случаев, когда вы будете трактовать значения как объекты, Java предоставляет стандартный класс-обертку для каждого простого типа, как показано в табл. 5.1.

Экземпляр класса-обертки воплощает единственную переменную его соответствующего типа. Это неизменяемый объект, который служит контейнером для хранения значения и позволяет нам получить его позже. Вы можете сконструировать объект-обертку из простого значения или представления значения в типе `String`.

Табл. 5.1. Обертки простых типов

Простой тип	Обертка
`void`	`java.lang.Void`
`boolean`	`java.lang.Boolean`
`char`	`java.lang.Character`
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

Следующие операторы равнозначны:

```
Float pi = new Float( 3.14 );
Float pi = new Float( "3.14" );
```

Конструкторы оберток выбрасывают исключение `NumberFormat Exception`, когда возникает ошибка в разборе строки.

Каждая обертка числового типа реализует интерфейс `java.lang.Number`, который предоставляет доступ-«значение» к методам его значения во всех простых формах. Вы можете получить скалярные значения при помощи методов `doubleValue()`, `floatValue()`, `longValue()`, `intValue()`, `shortValue()`, и `byteValue()`:

```
Double size = new Double ( 32.76 );
double d = size.doubleValue(); // 32.76
float f = size.floatValue(); // 32.76
long l = size.longValue(); // 32
int i = size.intValue(); // 32
```

Этот код равнозначен приведению типа double к различным типам.

Наиболее распространенное применение для оберток — это когда вы хотите передать простое значение методу, который нуждается в объекте. Например, в главе 11 мы рассмотрим API библиотек Java, изощренный набор классов для работы с группами объектов, такими как списки, наборы и карты. Все API библиотек работают с объектными типами, поэтому простые типы должны быть обернуты при хранении в них. В следующей главе мы увидим, что Java делает этот процесс оборачивания автоматическим. Однако сейчас давайте сделаем это сами. Как мы увидим, `List`

является расширяемой библиотекой представителей класса `Object`. Мы можем использовать обертки, чтобы хранить числа в `List` (вместе с другими объектами):

```
// Простой код Java
List myNumbers = new ArrayList();
Integer thirtyThree = new Integer( 33 );
myNumbers.add( thirtyThree );
```

Здесь мы создали объект-обертку класса `Integer`, так что можем вставить число в `List`, используя метод `add()`, который принимает объект. Позже, когда мы будем извлекать элементы из `List`, мы сможем восстановить значение `int` таким образом:

```
// простой код Java
Integer theNumber = (Integer)myNumbers.get(0);
int n = theNumber.intValue(); // 33
```

Как упоминалось ранее, когда мы позволяем Java делать это для нас, код становится более лаконичным и безопасным. Применение класса-обертки по большей части скрыто от нас компилятором, но она все еще используется внутренне:

```
// Java code, использующий автозапаковывание и обобщения
List<Integer> myNumbers = new ArrayList<Integer>();
myNumbers.add( 33 );
int n = myNumbers.get( 0 );
```

Этот пример будет иметь больший смысл, когда вы прочтете следующую главу об *автозапаковывании* и *распаковывании* простых значений.

Автоматическое упаковывание и распаковывание простых типов

Компилятор Java автоматически упаковывает простые числа в их типы оберток и распаковывает их, когда необходимо. Этот процесс называется «упаковывание и распаковывание простого числа». Он происходит, когда простые числа используются как аргументы и возвращают значения в методы и при простом назначении переменным. Например:

```
// простые назначения
Integer integer = 5;
int i = new Integer(5);
```

```
// Аргументы методов и типы возврата
Double multiply( Double a, Double b ) {
return a.doubleValue() * b.doubleValue();
}
double d = multiply( 5.0, 5.0 );
```

В первом случае Java просто обернул значение 5 в класс `Integer` для нас. Во втором случае он развернул наш объект `Integer` в его простое значение. Далее мы имеем метод, который умножает два объекта-упаковки `Double` и возвращает результат как обертку `Double`. Этот пример в действительности имеет три случая упаковки и один случай распаковки. Во-первых, два простых значения `double` оборачиваются в типы `Double`, чтобы вызвать метод. Далее оператор возврата метода на самом деле вызывается на простое значение `double`, которое компилятор превращает в `Double` до того, как оно покинет метод. В конце концов, компилятор распаковывает значение возврата во время назначения простой переменной `double d`.

Влияние упаковывания на производительность

Измерение производительности — непростая задача. Для подавляющего большинства приложений время, которое тратится на выполнение таких операций, как создание маленького объекта или вызов метода, является незначительным по сравнению с другими факторами, такими как ввод/вывод, взаимодействие пользователя или фактическая логика приложения. Как правило, не стоит переживать об этих проблемах в деталях выполнения, пока само приложение тщательно не обдумано. Однако мы можем предвидеть, что разрешение Java упаковывать и распаковывать простые типы в областях, критических для выполнения, будет не таким быстрым, как использование простых типов напрямую. Одним из аспектов, которые следует принять во внимание, является то, как много новых объектов создается и возвращается очисткой памяти. Если в основном Java можно заставить создать новый объект для каждого запакованного простого типа, существуют оптимизации для малого диапазона значений. Java гарантирует, что булевы значения `true` и `false`, как и числовые типы с «маленькими» значениями от 0 до 127 для байтов и символов и от –128 до 127 для коротких целых и целых, являются *изолированными*. Говоря, что они изолированы, мы имеем в виду, что вместо создания новых объектов каждый раз Java заново использует те же самые объекты при последующей упаковке. Это безопасно, поскольку упаковки простых типов являются постоянными и не могут изменяться.

```
Integer i = 4;
Integer j = 4;
System.out.println( i == j ); // Равенство этого объекта является истинным
только для маленьких значений
```

Результатом, как показано в этом куске кода, является то, что для маленьких сходных значений запакованные простые типы в действительности являются теми же объектами. Java также пытается интернировать строчные значения в классах Java. Мы поговорим об этом в главе 10.

Список аргументов переменной длины

Как мы упоминали ранее, методы Java могут иметь *список аргументов переменной длины*, или vararg, что позволяет им принимать любое количество аргументов при вызове. Самый распространенный пример использования `vararg` — это метод для вывода форматированного текста `printf()`, который позволяет вставлять любое количество тегов в строку и принимает аргумент для каждого тега для вывода. Например:

```
void printObjects( Object ... list ) {
// список является Object []
for( Object o : list )
System.out.println( o );
}
```

`Vararg` позволяет методу `printf()` принимать любое количество элементов для вывода (от ноля до дюжин, настолько громоздко, насколько это может быть громоздко).

Метод принятия списка аргумента переменной эквивалентен методу принятия массива некоего типа объекта. Разница заключается в том, что компилятор заставляет вызов метода принимать индивидуальные значения, разделенные запятой, а затем упаковывает их в массив для нас. Синтаксис объявления метода `varargs` использует многоточие (...), где должны быть квадратные скобки в массиве. Например:

```
void printObjects( Object ... list ) {
// список является Object []
for( Object o : list )
System.out.println( o );
}
```

Внутри метода `printObjects()` переменная `list` имеет в действительности тип `Object []`. Мы можем выяснить, сколько аргументов было передано нам, спросив массив о его длине обычным путем:

```
System.out.println( "Количество аргументов:" + list.length );
```

Если вызывающий оператор не передал ни одного аргумента, массив будет пустым. В случае нашего метода `printObjects()` мы могли передать смесь простых типов так же, как типов объектов, потому что компилятор автоматически упаковывает для нас простые типы в их типы оберток перед помещением их в `Object []`. Список переменных аргументов не должен иметь тип `Object`. Он может быть любого типа, включая простые типы. Например:

```
printInts( int ... list ) {
// список является int []
}
// использование
printInts( 1, 2, 3, 4 );
printStrings( String ... list ) {
// список является String []
}
// использование
printStrings( "foo", "bar", "gee" );
```

Метод `printInts()` получает массив `int []` значений простых целых чисел. Метод `printStrings()` получает в качестве аргумента `String []`. Фактические аргументы должны быть все назначаемыми (возможно, после числового продвижения или упаковывания) для типа списка переменных аргументов. Другими словами, метод `printInts()` может вызываться только при помощи классов `String`.

Методы `Vararg` могут также иметь любое количество фиксированных аргументов перед объявлением `vararg`. Таким образом метод `printf()` гарантирует, что его первый аргумент имеет формат строки.

```
void printf( String format, Object ... args ) { ... }
```

Перегрузка метода

Перегрузка — это возможность определить в одном классе несколько методов с одним и тем же именем; когда метод вызывается, компиля-

тор подбирает один, основанный на аргументе, переданном методу. Это предполагает, что перезагруженные методы должны иметь различное количество типов аргументов. (В главе 6 мы рассмотрим *перемещение метода*, которое возникает, когда мы объявляем методы с идентичными сигнатурами в разных классах.)

Перегрузка методов (также называемая *ситуативным полиморфизмом*) является мощным и полезным свойством. Идея заключается в создании методов, которые ведут себя аналогичным образом с различными типами аргументов. Это создает иллюзию, что один метод может оперировать многими типами аргументов. Метод `print()` в стандартном классе `PrintStream` является хорошим примером перегрузки метода в действии. Как вы, наверное, уже догадались, вы можете вывести почти все, что угодно, в форме строки, используя это выражение:

```
System.out.print( аргумент )
```

Переменная `out` является ссылкой на объект (`PrintStream`), который определяет девять различных «перегруженных» версий метода `print()`. Версии принимают аргументы следующих типов: `Object`, `String`, `char[]`, `char`, `int`, `long`, `float`, `double`, и `boolean`.

```
class PrintStream {
void print( Object arg ) { ... }
void print( String arg ) { ... }
void print( char [] arg ) { ... }
...
}
```

Вы можете вызвать метод `print()`, имея аргумент любого из этих типов, и он будет выводиться подходящим способом.

В языке без перегрузки методов это требует какого-то более громоздкого решения, как, например, уникально названный метод для вывода каждого типа объекта. В таком случае вашей обязанностью является выяснение того, какой метод используется для каждого типа данных.

В предыдущем примере `print()` был перегружен для поддержки двух ссылочных типов: `Object` и `String`. Что если мы попытаемся вызвать `print()` с каким-то другим ссылочным типом? Например, с объектом `Date`? Когда нет точного совпадения типа, компилятор ищет приемлемое, назначаемое *совпадение*. Поскольку `Date` является подклассом `Object`, как и все классы, объект `Date` может быть назначен переменной

типа `Object`. Таким образом, это является приемлемым совпадением и выбирается метод `Object`.

А что если у нас окажется сразу несколько совпадений? Например, мы пытаемся вывести подкласс `String`, называемый `MyString`. (Класс `String` является конечным, поэтому в действительности у него не может быть подклассов, но давайте используем наше воображение.) `MyString` назначается `String` или `Object`. Здесь компилятор определяет, какое совпадение «лучше», и выбирает этот метод. В данном случае это метод `String`.

Интуитивное объяснение заключается в том, что класс `String` «ближе» к `MyString` в иерархии наследования. Это более специфическое совпадение. Выражаясь точнее, данный метод *более специфичный*, чем другой метод, если типы аргументов первого метода все являются назначаемыми типам аргумента второго метода. В этом случае метод `String` более специфичен для подкласса `String`, чем метод `Object`, поскольку тип `String` назначаем типу `Object`. Обратное не является истиной.

Если вы внимательно следите, то, возможно, заметили: мы сказали, что компилятор разрешает перегруженные методы. Перегрузка метода — это не то, что происходит во время выполнения; это важное разграничение. Это означает, что выбранный метод выбирается однажды, когда код компилируется. Когда перегруженный метод выбран, выбор фиксируется, пока код не перекомпилируется, даже если класс, содержащий вызываемый метод, позже пересматривается и даже если добавляется более специфичный метод перегрузки. В этом отличие от *перемещения методов*, которые дислоцируются во время выполнения и могут быть найдены, даже если они не существовали, когда компилировался вызывающий класс. На практике это отличие обычно не будет важным для вас, поскольку вы, скорее всего, будете перекомпилировать все необходимые классы в одно и то же время. Мы поговорим о методах перемещения позже в этой главе.

Создание объектов

Объекты в Java размещаются в области памяти под названием «куча». В отличие от других языков мы не должны сами управлять этой памятью. Java заботится о выделении и распределении памяти для вас. Java эксплицитно выделяет хранилище для объекта, когда вы его создаете при помощи оператора `new`. Что более важно, объекты убираются сборщиком мусора, когда на них больше не ссылаются.

Конструкторы

Объекты создаются оператором `new` при использовании *конструктора объектов*. Конструктор — это специальный метод с тем же именем, что и у его класса, и без возвращаемого типа. Он вызывается, когда создается новый экземпляр класса, что дает классу возможность устанавливать объект для использования. Конструкторы, как другие методы, могут принимать аргументы и могут быть перегружены (однако они не наследуются, как другие методы; мы исследуем наследование в главе 6).

```
class Date {
long time;
Date() {
time = currentTime();
}
Date( String date ) {
time = parseDate( date );
}
...
}
```

В этом примере у класса `Date` два конструктора. У первого нет никаких аргументов; он известен как *конструктор по умолчанию*. Конструкторы по умолчанию играют особую роль: если мы не определим никакого конструктора для класса, нам предоставляется пустой конструктор, который используется по умолчанию. Он вызывается, когда вы создаете объект, вызывая его конструктор без аргумента. Здесь мы применили дефолтный конструктор, так что он устанавливает переменную экземпляра `time`, вызывая гипотетический метод `currentTime()`, который соответствует функциональности реального класса `java.util.Date`. Второй конструктор принимает аргумент класса `String`. Предположительно этот `String` содержит строчное представление анализируемого времени, чтобы установить переменную `time`. Принимая во внимание конструкторы из предыдущего примера, мы создаем объект класса `Date` следующим образом:

```
Date now = new Date();
Date christmas = new Date("Янв 07, 2006");
```

В каждом случае Java во время компиляции выбирает подходящий конструктор, основанный на правилах для выбора перегруженных методов.

Если позже мы уберем все ссылки на назначенный объект, он очистится из памяти, как мы увидим далее:

```
christmas = null; // честная игра для очистки памяти
```

Установка значения ссылки равным `null`, означает, что она больше не указывает на строчный объект "Янв 07, 2006". Присвоив переменной `christmas` любое другое значение, мы бы добились того же эффекта.

Если другая переменная не ссылается на исходный строчный объект, она будет недоступной и может быть удалена из памяти. Мы не советуем вам назначать ссылке значение `null`, чтобы очистить значения из памяти. Часто это просто происходит само собой, когда локальные переменные выходят за границы, но элементы, на которые ссылаются переменные экземпляра объектов, существуют столько же, сколько существует объект (через ссылки на него), а статические переменные существуют по сути вечно.

Еще несколько примечаний: конструкторы не могут быть объявлены как `abstract`, `synchronized` или `final` (мы определим остальные термины позже). Конструкторы могут, однако, быть объявлены при помощи модификаторов видимости `public`, `private` или `protected`, как другие методы, контролирующие их доступность. Мы поговорим в подробностях о модификаторах видимости в следующей главе.

Работа с перегруженными конструкторами

Конструктор может ссылаться на другой конструктор в том же или родительском классе, используя специальные формы `this` и суперссылки. Мы обсудим первый случай и вернемся к конструктору родительского класса после того, как еще поговорим о подклассах и наследовании. Конструктор может вызвать другой перегруженный конструктор в его классе, передавая подходящие аргументы методу `this()`, который ссылается на текущий класс, чтобы выбрать желаемый конструктор. Если конструктор вызывает другой конструктор, *он должен делать это как его первый оператор*:

```
class Car {
String model;
int doors;
Car( String model, int doors ) {
this.model = model;
```

```
this.doors = doors;
// другая, сложная установка
...
}
Car( String model ) {
this( model, 4 /* двери */ );
}
...
}
```

В этом примере у класса `Car` два конструктора. Первый, более конкретный, принимает аргументы, указывающие модель машины и количество ее дверей. Второй конструктор берет в качестве аргумента только модель и, в свою очередь, вызывает первый конструктор с дефолтным значением четырех дверей. Преимущество этого подхода заключается в том, что один конструктор может сделать всю сложную работу по установке; другие вспомогательные конструкторы просто скармливают подходящие аргументы этому конструктору.

Вызов специального метода `this()` должен размещаться в самом начале нашего конструктора. Синтаксис в этом способе ограничен, потому что существует потребность в определении четкой цепочки команд в вызове конструктора. В конце цепочки Java вызывает конструктор надкласса (если мы не сделаем это явно), чтобы обеспечить правильную инициализацию унаследованных методов перед тем, как мы продолжим.

Есть также момент в цепочке сразу после вызова конструктора надкласса, где вычисляются инициализаторы экземпляров переменных текущего класса. До этой точки мы даже не можем сослаться на переменные экземпляра нашего класса. Мы объясним эту ситуацию более полно после того, как поговорим о наследовании.

Сейчас все, что вам нужно знать, — это то, что вы можете вызвать второй конструктор (передать ему) как первый оператор вашего конструктора. Например, следующий код является неправомерным и вызывает ошибку во время компиляции:

```
Car( String m ) {
int doors = determineDoors();
this( m, doors ); // Ошибка: вызов конструктора
// должен быть первым оператором
}
```

Простой конструктор, который принимает название модели автомобиля, не может производить дополнительную установку перед вызовом более явного конструктора. Он даже не может сослаться на член экземпляра для постоянного значения:

```
class Car {
...
final int default_doors = 4;
...
Car( String m ) {
this( m, default_doors ); // Ошибка: ссылка
// на неинициализированную переменную
}
...
}
```

Переменная экземпляра `defaultDoors` не инициализирована, пока последний пункт в цепи в конструкторе вызывает установку объекта, поэтому компилятор не позволяет нам получить доступ к ней. К счастью, мы можем решить эту отдельную проблему при использовании статической переменной вместо переменной экземпляра:

```
class Car {
...
static final int DEFAULT_DOORS = 4;
...
Car( String m ) {
this( m, DEFAULT_DOORS ); // Ок!
}
...
}
```

Статические члены класса инициализируются, когда класс впервые загружается в виртуальную машину, поэтому безопасно получить к ним доступ из конструктора.

Статические и нестатические блоки инициализаторов

Можно объявить блок кода (некоторые операторы в фигурных скобках) непосредственно в рамках класса. Этот блок кода не принадлежит никакому методу; вместо этого он выполняется всего один раз — при

создании или в случае использования модификатора static, загрузке класса. Эти блоки могут быть использованы для произведения дополнительной подготовки класса или экземпляра объекта и называются *блоки инициализаторов*.

Блоки инициализаторов экземпляра могут восприниматься как инициализация переменной экземпляра. Они вызываются в то время, когда вычисляются инициализаторы переменной экземпляра (после конструкции надкласса, но перед телом вашего конструктора), в том порядке, в котором они возникают в коде Java:

```
class MyClass {
Properties myProps = new Properties();
// установить myProps
{
myProps.put("foo", "bar");
myProps.put("boo", "gee");
}
int a = 5;
...
```

Обычно этот вид установки может быть также произведен в конструкторе объекта. Примечательным исключением является случай с анонимным внутренним классом (см. главу 6).

Подобным образом вы можете использовать статические инициализаторы блока, чтобы установить статические члены класса.

Этот более полезный случай позволяет статическим членам класса иметь сложную инициализацию, так же как поступают объекты с конструкторами:

```
class ColorWheel {
static Hashtable colors = new Hashtable();
// установить цвета
static {
colors.put("Red", Color.red );
colors.put("Green", Color.green );
colors.put("Blue", Color.blue );
...
}
...
}
```

Класс `ColorWheel` предоставляет переменную `colors`, которая обозначает названия цветов в объектах `Color` в `Hashtable`. Когда впервые ссылаются на класс `ColorWheel` и когда он впервые загружается, статические компоненты `ColorWheel` вычисляются в том порядке, в котором они появляются в источнике. В этом случае блок статического кода просто добавляет элементы к таблице `colors`.

Разрушение объектов

Теперь, когда мы увидели, как создать объекты, пришло время поговорить об их разрушении. Если вы привыкли к программированию на C и C++, вы, возможно, провели много времени, выслеживая утечку памяти в вашем коде. Язык Java берет на себя ответственность за разрушение объекта; вам не нужно волноваться о традиционной утечке памяти, и вы можете сосредоточиться на более важных задачах программирования*.

Сборка мусора

Язык Java использует технологию, известную как *сборка мусора*, чтобы убрать объекты, которые больше не нужны. Программа сбора мусора в Java является «беспощадным жнецом». Она работает в фоновом режиме, выслеживая объекты и ожидая их кончины. Она находит их и следит за ними, периодически считая ссылки на них, чтобы определить, когда пришло их время. Когда все ссылки на объект пропали и более не доступны, механизм очистки памяти объявляет объект *недоступным* и восстанавливает место, которое он занимал в доступном накопителе ресурсов. Недоступный объект — это объект, который больше нельзя найти через комбинацию «живых» ссылок в работающем приложении.

Сборка мусора использует разнообразие алгоритмов; архитектура виртуальной машины Java не предусматривает какой-то конкретной методики. Однако то, как некоторые реализации в Java выполняли эту задачу, заслуживает упоминания. Вначале в Java использовалась технология, называемая «отслеживание и очистка». Согласно этой схеме Java сначала проходит по дереву всех доступных ссылок объектов и помечает их как

---

* В Java до сих пор можно писать код, который удерживает объект вечно, потребляя все больше и больше памяти. Это не столько утечка, сколько накопление памяти. Также это обычно намного легче отследить, используя правильные методики и инструменты.

живые. Затем Java сканирует динамическую память, выискивая идентифицируемые объекты, которые не были помечены. В этой технике язык Java способен найти объекты в динамической памяти, поскольку они хранятся характерным путем и имеют определенную битовую подпись в их описателях, которую нельзя воспроизвести естественным образом. Этот вид алгоритма не запутывает проблема циклических ссылок, в которой объекты могут взаимно ссылаться друг на друга и оказываться действующими, даже если они мертвы (Java обрабатывает эту проблему автоматически). Однако эта схема была не самым быстрым методом и вызывала паузы в программе. С тех пор реализации стали намного более сложными.

Современные программы сборки мусора эффективно работают непрерывно, не вызывая длительной задержки в выполнении приложения Java. Поскольку они являются частью системы выполнения, они также могут выполнять некоторые вещи, которые нельзя делать статически. Реализация Java от Sun делит динамическую память на несколько областей для объектов с по-разному оцененной длительностью жизни. Объекты с короткой длительностью жизни помещаются в специальную область динамической памяти, которая уменьшает время их решительного сброса. Объекты, которые живут дольше, могут быть перемещены в другие, менее изменчивые части динамической памяти. В последних реализациях программа очистки памяти может даже настроить себя, регулируя размер частей динамической памяти, на основе фактического выполнения приложения. Улучшения в программе очистки памяти со времени ранних релизов весьма значительны, они стали одной из причин того, что язык Java теперь почти эквивалентен в скорости традиционным компилированным языкам.

В основном вам не следует беспокоиться о процессе очистки памяти. Но один метод очистки памяти может быть полезен для отладки. Вы можете указать программе очистки памяти на произведение чистки явно, вызвав метод `System.gc()`. Этот метод является полностью зависимым от реализации и может ничего не сделать, но его можно использовать, если вы хотите получить гарантию того, что Java очистилась перед совершением какого-то действия.

Завершение

Перед тем как объект будет устранен программой очистки памяти, вызывается его метод `finalize()`, чтобы дать ему последнюю возможность хотя бы закончить свое действие и освободить другие типы ресур-

сов, которые он может удерживать. Поскольку ваша программа очистки памяти способна восстанавливать ресурсы памяти, она может заботиться о таких вещах, как закрытие файлов и прерывание сетевых соединений, не столь элегантно и эффективно как ваш код. Именно для этого нужен метод `finalize()`. Метод объекта `finalize()` вызывается однажды и только однажды перед тем, как объект очистится из памяти. Однако нет гарантии того, когда это случится. Очистка памяти может в теории никогда не запуститься в системе, в которой нет недостатка памяти. Также интересно подметить, что финализация и очистка осуществляется за две особые фазы в процессе очистки памяти. Сначала элементы финализируются, затем они очищаются. Следовательно, есть вероятность того, что финализация может (намерено или непреднамеренно) создать замедленную ссылку на объект, о котором идет речь, откладывая его очистку из памяти. Объект, конечно, подлежит очистке в дальнейшем, если ссылка исчезнет, но его объект `finalize()` не вызывается снова.

Методы `finalize()` надклассов не вызываются для вас автоматически. Если вам нужно вызвать рутинную программу ваших родительских классов, вы можете вызвать метод `finalize()` вашего надкласса, используя `super.finalize()`. Мы обсудим наследование и перезаписываемые методы в главе 6.

Слабые и мягкие ссылки

В основном, как мы описали, программа очистки памяти Java утилизирует объекты, когда они недоступны. Недоступный объект, опять же, это тот объект, на который не ссылается ни одна переменная в вашем приложении и который недоступен через какую-либо цепь ссылок никакому работающему потоку. Такой объект больше не может быть использован никаким приложением и, таким образом, является несомненным случаем, когда объект должен быть удален.

Однако, в некоторых ситуациях выгодно, чтобы программа очистки памяти Java работала с вашим приложением и решала, когда настанет время удалить отдельно взятый объект. Для таких случаев Java позволяет вам хранить ссылку на объект неявно посредством специального объекта-оболочки типа `java.lang.ref.Reference`. Если Java затем решит удалить объект, ссылка, которую хранит обертка, автоматически превратится в `null`. Пока ссылка существует, вы можете продолжать использовать ее обычным образом и назначать ее куда угодно (используя обычные ссылки), предотвращая очистку памяти.

Существует два типа оберток `Reference`, которые реализуют разные схемы для ситуации, когда нужно очистить память от целевых ссылок. Первый называется `WeakReference`.

Слабые ссылки сразу подходят для немедленной очистки памяти; они не предотвращают очистку памяти, как делают обычные «сильные» ссылки. Это означает, что если у вас в приложении существует комбинация сильных ссылок и ссылок, содержащихся в обертках `WeakReference`, программа очистки памяти ожидает, пока останутся только обертки `WeakReference`, и затем соберет объекты. Это существенная особенность, которая позволяет программе очистки памяти работать с определенными типами схем кеширования. Часто вы захотите кешировать ссылку на объект для выполнения (чтобы избежать ее создания или для ее поиска). Но пока вы не предпримете особые действия, чтобы устранить ненужный объект из кеша, кеш будет всегда хранить ваши объекты, сохраняя действующие ссылки на них. Используя слабые ссылки, вы можете реализовать кеш, который автоматически выбрасывает ссылки, когда объект автоматически должен быть выброшен из памяти. В действительности считается, что реализация `HashMap`, вызванная `WeakHashMap`, делает только это (см. подробности в главе 11).

Второй тип обертки ссылок называется `SoftReference`. *Мягкая ссылка* похожа на слабую ссылку, но она говорит программе очистки памяти быть менее агрессивной, утилизируя его содержимое. Объекты с мягкими ссылками очищаются, только когда и если у Java недостаток памяти. Это полезно для немного другого типа кеширования, где вы захотите сохранить какое-то содержимое, если нет потребности избавиться от него. Например, веб-браузер может использовать мягкие ссылки, чтобы кешировать изображения или строки HTML изнутри, таким образом, оставляя их так долго, как это возможно, пока не вступят в игру ограничения памяти. (Более сложное приложение может также использовать собственную схему, основанную на пометке «наименее часто используемые» какого-то рода.)

Пакет `java.lang.ref` содержит обертки `WeakReference` и `SoftReference`, как и способность, называемую `ReferenceQueue` и позволяющую вашему приложению получить список ссылок, которые были собраны. Важно, что ваше приложение использует очередность или какой-то другой механизм, чтобы удалить сами объекты `Reference` после того, как их содержимое собрано; в другом случае кеш скоро заполнится пустыми обертками объекта `Reference`.

Перечисления

Сейчас, когда мы освоили базовую информацию о классах, мы можем поговорить более подробно о *перечислениях*. Как мы уже сказали, перечисление — это тип объекта в языке Java, который ограничен определенным набором значений. Порядок следования значений зависит от порядка их объявления в коде; сами значения соответствуют строковым именам, которые были указаны при объявлении.

Мы уже увидели пару примеров перечисления, использующихся вместо статических идентификаторов. Например:

```
enum Weekday { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday }
// использование
setDay( Weekday.Sunday );
```

Давайте посмотрим на то, что в действительности генерирует компилятор Java для `enum`. Это регулярный компилируемый класс Java, в этом случае названный `Weekday`, так что можем показать его командой `javap` таким образом:

```
% javap Weekday
public final class Weekday extends java.lang.Enum {
public static final Weekday Sunday;
public static final Weekday Monday;
public static final Weekday Tuesday;
public static final Weekday Wednesday;
public static final Weekday Thursday;
public static final Weekday Friday;
public static final Weekday Saturday;
public static final Weekday[] values();
public static Weekday valueOf(java.lang.String);
}
```

Класс `Weekday` является подклассом типа перечислений `Enum` с семью статическими, финальными, «константными» ссылками объектов, соответствующих нашим пронумерованным значениям. Каждое из пронумерованных значений имеет тип `Weekday`. Компилятор Java не позволяет нам расширить этот класс или создать какой-либо другой экземпляр этого типа. Единственные экземпляры `Weekday`, которые когда-либо будут существовать, это семь пронумерованных значений. Это то, что

обеспечивает перечислениям безопасность типа. Методу, ожидающему класс `Weekday`, может быть передано только одно из семи значений. В отличие от числового константного идентификатора, никакое значение, отличное от `Weekday`, не сработает. Как мы увидели в главе 4, перечисления (в отличие от большинства объектов) могут также использоваться с операторами `switch` с тем же преимуществом.

Поскольку перечисления являются статическими значениями, они могут быть импортированы при помощи Java-оператора `static import`, сохраняя для нас некоторые типы.

```
import static mypackage.Weekday.*;
...
setDay( Friday );
setDeadline( Sunday );
```

Мы также должны упомянуть, что перечисления могут быть вызваны только вдобавок к классам «верхнего уровня», но внутри классов и ссылок. В этом случае они ведут себя как внутренние классы (см. главу 6).

Значения класса `Enum`

Вы можете получить упорядоченный список значений перечисления при помощи статического метода `values()`.

```
Weekday [] weekdays = Weekday.values();
```

Метод перечисления `compareTo()` сравнивает одно значение перечисления с другим значением того же типа перечисления и возвращает целое число меньше ноля, ноль или больше ноля, указывая, является ли целевое перечисление «меньше, чем», «равно» или «больше, чем» порядок перечисляемых ссылок. Это не очень важно для нашего `Weekdays`, но может быть полезно для значений, которые имеют более числовое значение или (нециклическую) величину какого-то типа. Например:

```
Level level = Level.LOW;

Level anotherLevel = Level.HIGH;

if ( level.compareTo( anotherLevel ) > 0 ) // истина

doSomething();
```

Мы упоминали, что перечисляемые значения имеют строчное соответствие их именам. Вы можете получить строчное имя значения (которое является точно таким же, как заявлено в исходном коде) с методом `name()`. Идя в другом направлении, вы можете «отыскать» любое значение перечисления по его типу класса или строчному имени, используя статический метод `Enum.valueOf()`:

```
String mondayString = Weekday.Monday.name(); // "Понедельник"
Weekday mondayWeekday = Enum.valueOf( Weekday.class, "Понедельник" );
```

Значение метода `name()` также используется методом значения `toString()`, поэтому вывод значения перечисления делает то, что вы ожидаете.

Изменение перечислений

Мы сказали, что тип `java.lang.Enum` не может быть непосредственно расширен и что вы не можете создать новые экземпляры типов `enum`. Однако вы можете добавлять элементы сгенерированному классу перечисления, когда он объявлен. Например, перечисление `java.util.concurrent.TimeUnit`, которое имеет идентификаторы для таких единиц времени, как `SECONDS`, `MILLISECONDS` и `MICROSECONDS`, имеет метод `sleep()`, который интерпретирует его аргументы в правильную временную шкалу:

```
import static java.util.concurrent.TimeUnit.*;
SECONDS.sleep( 5 ); // спать 5 секунд
```

Перечисления могут иметь значения с конструкторами, методами и полями, как другие классы. По большей части это просто; вы просто добавляете точку с запятой после значений перечисления, а затем добавляете ваши дополнительные члены класса. Давайте добавим значение `fun` и метод доступа нашим дням недели:

```
public enum Weekday
{
Sunday(8), Monday(0), Tuesday(1), Wednesday(2), Thursday(4),
Friday(6), Saturday(10) ;
int fun;
Weekday( int fun ) { this.fun = fun; }
public int getFun() { return fun; }
}
```

Здесь мы добавили переменную экземпляра `fun` классу `Weekday`, а также метод конструктора и метод доступа, который работает со значением. Объявление каждого из значений перечисления теперь принимает значение конструктора подобно вызову метода без ключевого слова `new`. Отметьте, что точка с запятой в конце значений является обязательной. Теперь каждый `Weekday` имеет атрибут `fun`.

Есть одна нетипичная специфическая черта перечислений, которую мы не показали. В дополнение к добавлению свойств всему классу перечисления (как в нашем примере) мы можем добавлять методы и переменные отдельным значениям перечисления, давая им тело в фигурных скобках (`{}`). Лучше всего это демонстрирует пример:

```
enum Cat {
Himilayan, Siamese, Caleco,
Persian {
public void someMethod() { ... }
}
}
```

Теперь только значение перечисления `Cat.Persian` имеет метод. В этом случае компилятор генерирует подкласс `Cat` как внутренний класс типа `Persian` для хранения дополнительного члена. (Мы поговорим о внутренних классах в главе 6.) Вы можете использовать это для перезаписи членом `Persian` метода в базовом классе перечисления.

Материал, защищенный авторским правом

Глава 6
ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

Мы уже увидели, как создавать в Java классы и объекты, которые являются экземплярами этих классов. Сами по себе классы — это нечто большее, чем просто средство организации кода. Именно в отношениях между объектами — их правах доступа и связях друг с другом — действительно выражается мощность объектно-ориентированного языка. Именно это мы рассмотрим в данной главе. В частности мы рассмотрим несколько видов отношений.

Отношения наследования

То, как класс наследует методы и переменные у родительского класса.

Интерфейсы

Как заявить то, что класс реализует определенное поведение и определяет тип, чтобы сослаться на это поведение.

Упаковка

Как организовать объекты в локальные группы.

Внутренние классы

Обобщение классов, которое позволяет вам вкладывать определение класса в другое определение класса.

Создание подклассов и наследование

Классы в Java существуют иерархически. Класс в Java может быть объявлен *подклассом* другого класса при использовании ключевого слова `extends`.

Подкласс *наследует* переменные и методы своего *родителя* и может использовать их, как если бы они были объявлены внутри самого подкласса:

```
class Animal {
float weight;
...
void eat() {
...
}
...
}
class Mammal extends Animal {
// наследует weight
int heartRate;
...
// наследует eat()
void breathe() {
...
}
}
```

В этом примере объект типа `Mammal` содержит экземпляры переменной `weight` и метода `eat()`. Они наследуются от `Animal`.

Класс может *расширять* только один другой класс. Используя правильную терминологию, скажем, что Java позволяет *единственное наследование* реализации класса.

Далее в этой главе мы поговорим об интерфейсах, которые замещают *множественное наследование*, преимущественно применяемое в других языках.

Можно создавать подклассы подклассов. Обычно подклассы делают классы более определенными или улучшают их, добавляя переменные и методы (вы не можете убрать или спрятать переменные или методы при помощи создания подклассов). Например:

```
class Cat extends Mammal {
// наследует weight и heartRate
boolean longHair;
...
// наследует eat() и breathe()
void purr() {
...
}
}
```

Класс `Cat` относится к типу `Mammal`, который наконец является типом `Animal`. Объекты `Cat` наследуют все характеристики объектов `Mammal` и, в свою очередь, объектов `Animal`. `Cat` также предоставляет дополнительное поведение в форме метода `purr()` и переменной `longHair`. Мы можем обозначить отношения класса при помощи диаграммы, как показано на рис. 6.1.

Подкласс наследует все члены надкласса как `private`. Как мы вскоре убедимся, другие уровни видимости влияют на то, какие унаследованные члены класса могут быть видны вне класса и его подклассов, но как минимум подкласс всегда имеет тот же набор видимых членов, что и его родитель. По этой причине тип *подкласса* может считаться подтипом его родителя, а экземпляры подтипа могут использоваться везде, где разрешены экземпляры супертипа. Рассмотрите следующий пример:

```
Cat simon = new Cat();
Animal creature = simon;
```

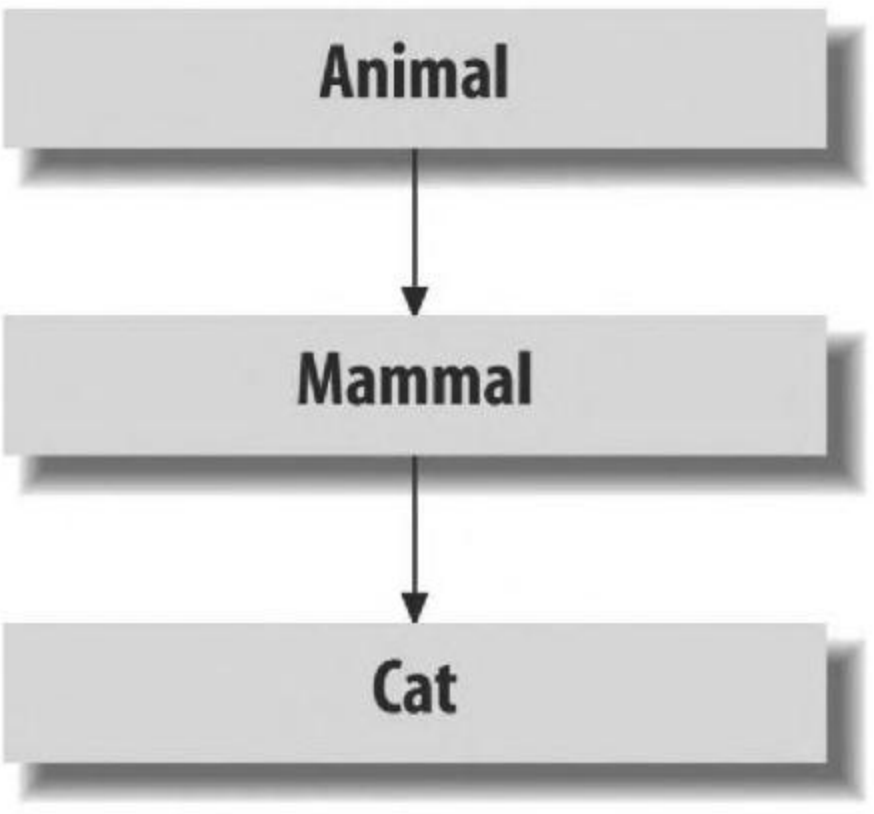

Рис. 6.1. Иерархия класса

Экземпляр `simon` класса `Cat` в этом примере может быть назначен переменной типа `creature` класса `Animal`, потому что `Cat` является подтипом `Animal`. Подобным образом любой метод, принимающий объект `Animal`, примет также экземпляр класса `Cat` или типа `Mammal`. Это является важным аспектом полиморфизма в таком объектно-ориентированном языке, как Java. Мы посмотрим, как это можно использовать, чтобы улучшить объект класса, а также добавить ему новые возможности.

Перекрытие переменных

В главе 5 мы рассмотрели, что локальная переменная, которая имеет такое же имя, что и переменная экземпляра, *перекрывает* (прячет) переменную экземпляра. Подобным образом переменная экземпляра в подклассе может скрывать переменную экземпляра с тем же именем, что и родительский класс, как показано на рис. 6.2. Мы рассмотрим в деталях укрытие этой переменной для полноты картины и для подготовки к более продвинутым темам, но на практике вы почти никогда не должны этого делать. Намного лучше структурировать ваш код, чтобы четко дифференцировать переменные, используя разные имена или соглашения об универсальных именах.

На рис. 6.2 переменная `weight` объявляется в трех местах: как локальная переменная в методе `foodConsumption()` класса `Mammal`, как переменная экземпляра класса `Mammal` и как переменная экземпляра класса `Animal`. Фактическая переменная, выбранная, когда вы ссылаетесь на нее в коде, зависит от границ, в которых мы работаем, и того, как вы определяете ссылку к ней.

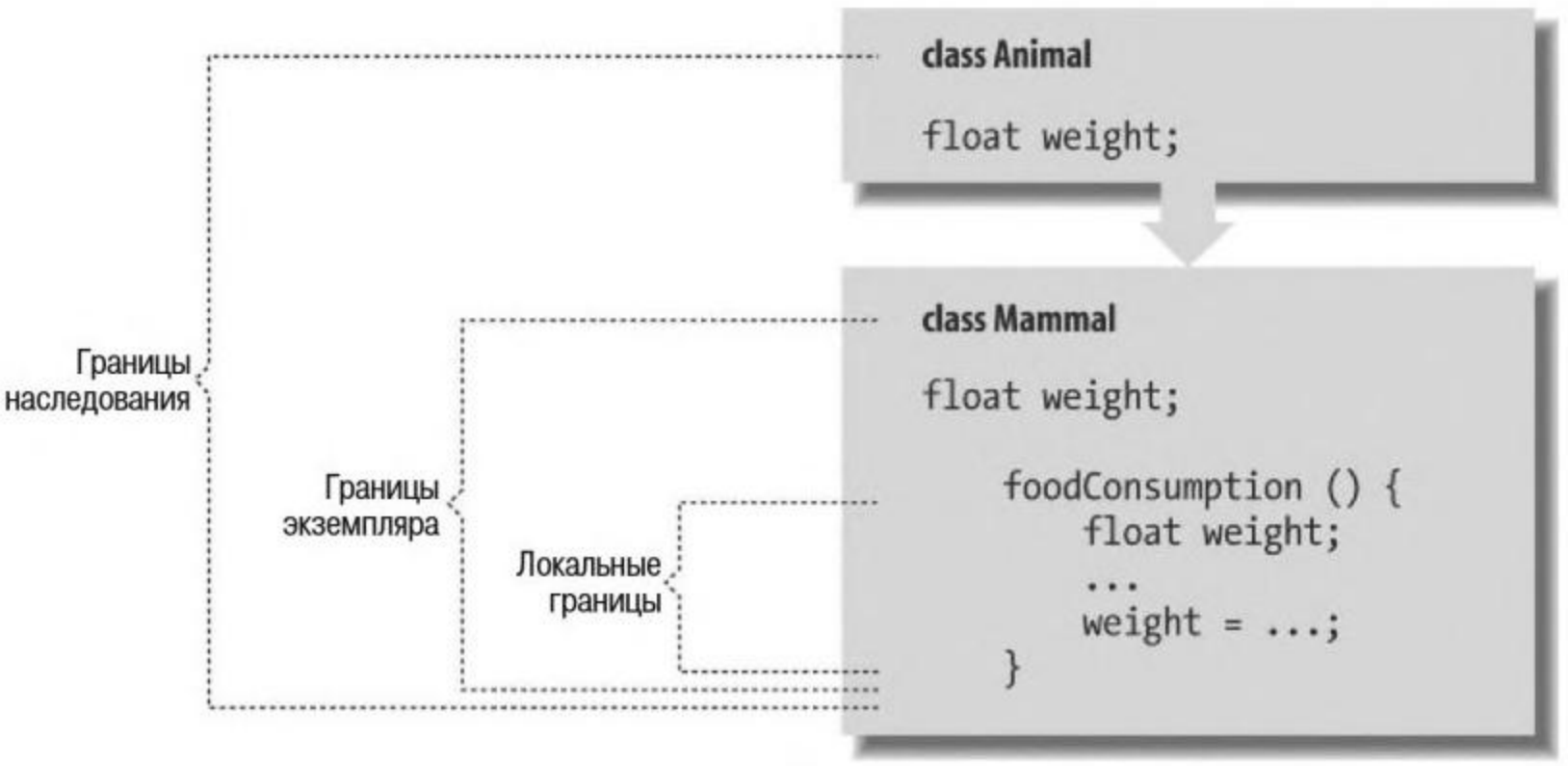

Рис. 6.2. Границы скрытой переменной

В предыдущем примере все переменные были одного типа. Более предпочтительный способ использования перекрытых переменных предполагал бы изменение их типов. Например, мы могли бы скрыть переменную `int` при помощи переменной `double` в подклассе, который нуждается в десятичных значениях вместо целочисленных значений. Мы можем делать это, не меняя существующий код, поскольку, как

предполагает ее имя, когда мы скрываем переменные, мы не замещаем их, а маскируем. Обе переменные до сих пор существуют; методы надкласса видят исходную переменную, а методы подкласса видят новую версию. Определение того, какие переменные видят различные методы, происходит во время компиляции.

Ниже представлен пример:

```
class IntegerCalculator {
int sum;
...
}
class DecimalCalculator extends IntegerCalculator {
double sum;
...
}
```

В этом примере мы скрываем переменную экземпляра `sum`, чтобы изменить ее тип с `int` на `double`*. Методы, определенные в классе `IntegerCalculator`, видят целую переменную `sum`, в то время как методы, определенные в `DecimalCalculator`, видят переменную `sum` с плавающей запятой. Однако обе переменные действительно существуют для данного экземпляра `DecimalCalculator`, и они могут быть независимыми значениями. В действительности любые методы, которые наследует `DecimalCalculator` от `IntegerCalculator`, фактически видят целую переменную `sum`.

Поскольку обе переменные существуют в `DecimalCalculator`, нам нужен способ, чтобы сослаться на переменную, унаследованную от `IntegerCalculator`. Мы делаем это, используя ключевое слово `super` в качестве спецификатора ссылки:

```
int s = super.sum;
```

Внутри `DecimalCalculator` ключевое слово `super`, использованное на такой же манер, выбирает переменную `sum`, определенную в надклассе. Вскоре мы объясним использование `super` более подробно.

Другим важным пунктом о скрытых переменных является то, как они работают, когда мы ссылаемся на объект посредством менее произ-

---

* Обратите внимание, что лучшим способом сконструировать наши вычисления будет создать абстрактный класс `Calculator` с двумя подклассами : `IntegerCalculator` и `DecimalCalculator`.

водного типа (родительского типа). Например, мы можем ссылаться на объект `DecimalCalculator` как на `IntegerCalculator`, используя его через переменную типа ntegerCalculator. Если мы так сделаем, а затем обратимся к переменной sum, мы получим целочисленную переменную, а не объект типа `DecimalCalculator`:

```
DecimalCalculator dc = new DecimalCalculator();
IntegerCalculator ic = dc;
int s = ic.sum; // обращается к sum класса IntegerCalculator
```

То же самое бы случилось, если бы мы обратились к объекту, использующему явное приведение к типу `IntegerCalculator` или при передаче экземпляра в метод, который принимает тот родительский тип.

Напоминаем, полезность скрытых переменных ограничена. Намного лучше отделять использование таких переменных другими способами, чем использовать сложные правила видимости. Однако важно понять концепцию перед тем, как мы начнем говорить о том же применительно к методам. Мы увидим другой и более динамический тип поведения, когда методы скрывают другие методы, или, если использовать правильную терминологию, *перезаписывают* другие методы.

Переопределение методов

В главе 5 мы увидели, что можем объявлять *перегруженные методы* (то есть методы с тем же именем, но другим количеством или типом аргументов) внутри класса.

Выбор перегруженного метода работает таким же образом, как мы обсуждали, со всеми методами, доступными классу, включая унаследованные. Это означает, что подкласс может определить дополнительные перегруженные методы, которые добавляются к перегруженным методам, предоставленным надклассом.

Подкласс может делать большее; он может определять метод, который имеет *точно такую же* сигнатуру (имя и тип аргументов), что и метод в родительском классе. В этом случае метод в подклассе *переписывает* метод родителя и эффективно замещает его реализацию, как показано на рис. 6.3. Переопределение методов для замены поведения объектов называется *полиморфизмом подтипов*. Это то решение, о котором думает большинство людей, когда они говорят о силе объектно-ориентированных языков.

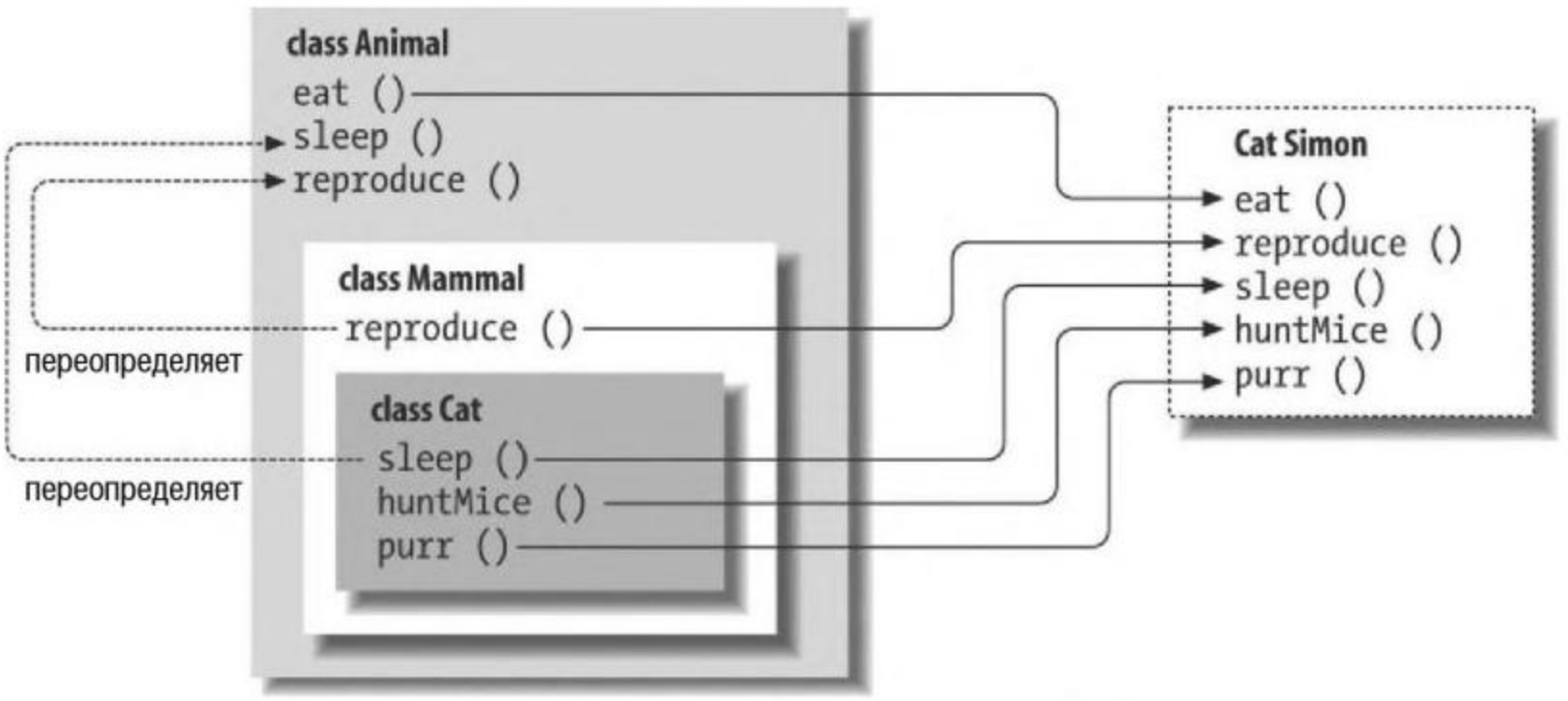

Рис. 6.3. Переопределение метода

На рис. 6.3 класс `Mammal` переопределяет метод `reproduce()` класса `Animal`, возможно, чтобы специализировать метод для поведения для млекопитающих*. Поведение сна объекта `Cat` также переопределяется, чтобы быть отличным от общего поведения представителей класса `Animal`, возможно, чтобы приспособить короткий сон. Класс `Cat` также добавляет более уникальный характер мурлыканья и охоты на мышей.

Из того, что вы уже видели, может сложиться впечатление, будто переопределенные методы скрывают методы в надклассах, также как переменные. Но переопределенные методы являются в действительности более мощными. Когда в иерархии наследования объекта множество реализаций метода, метод в «самом унаследованном» классе (самый дальний в иерархии) всегда переопределяет другие, даже если мы ссылаемся к объекту через ссылку одного из типов надкласса**.

Например, если экземпляр класса `Cat` назначен переменной более обобщенного типа `Animal`, и мы вызываем его метод `sleep()`, мы все еще получаем метод `sleep()`, реализованный в классе `Cat`, а не в классе `Animal`:

```
Cat simon = new Cat();
Animal creature = simon;
...
creature.sleep(); // обращается к Cat sleep();
```

---

* Утконос (`Platypus`) — очень необычное яйцекладущее млекопитающее (`Mammal`). Мы можем снова переопределить поведение `reproduce()` для него в его собственный подкласс `Mammal`.

** Переопределенный метод в Java ведет себя как метод `virtual` в языке C++.

Другими словами, для целей поведения (вызов методов), `Cat` ведет себя как `Cat`, независимо от того, ссылаетесь ли вы на него на как таковой. В остальном переменная `creature` здесь может вести себя как ссылка на `Animal`. Как мы объяснили ранее, обращение к скрытой переменной через ссылку `Animal` найдет реализацию класса `Animal`, а не класса `Cat`. Однако, поскольку методы расположены *динамически* при начальном поиске подклассов, вызывается подходящий метод в классе `Cat`, даже если мы обращаемся к нему в более общем смысле, как к объекту `Animal`. Это означает, что *поведение* объекта является динамическим. Мы можем относиться к специализированным объектам как к более общим типам и при этом получать выгоду от их специализированной реализации поведения.

`@Override`

Обычная ошибка программирования в языке Java — это случайная перегрузка метода при попытке переопределить его. Любое отличие в количестве или типе аргументов (*сигнатура* метода) производит два перегруженных метода вместо одного переопределенного. Новый синтаксис аннотаций в Java 5.0 дает возможность заставить компилятор помочь с этой проблемой. Аннотация, как мы обсудим в главе 7, позволяет нам добавить к источнику кода специальные ярлыки или метаданные, которые могут быть прочтены компилятором или инструментами выполнения. Одна из стандартных аннотаций, определяемая Java, называется `@Override`, и она говорит компилятору, что метод, который она помечает, предполагает переопределение метода в надклассе. Компилятор затем предупреждает, если метод не совпадает. Например, мы можем указать, что метод `sleep()` нашего класса `Cat` переопределяет метод в классе таким образом:

```
class Cat extends Mammal {
...
@Override void sleep() { ... }
}
```

Переопределенные методы и динамическое связывание

В предыдущем разделе мы упомянули, что *перегруженные* методы выбираются компилятором во время компиляции. С другой стороны, *переопределенные* методы выбираются динамически во время выполнения. Даже если мы создаем экземпляр подкласса, который никогда ранее не видел наш код (возможно, новый класс, загруженный через сеть), любые переопределяющие методы, которые он содержит, расположены

и используются во время выполнения, замещая те, которые существовали во время последней компиляции нашего кода.

Напротив, если мы создали новый класс, который применяет дополнительный, более специфический перегруженный метод, и замещает скомпилированные классы в нашем пути класса им, наш код продолжит использовать реализацию, которую он обнаружил изначально. Эта ситуация будет продолжаться, пока мы не перекомпилируем наш код новым классом. Другим эффектом этого является то, что приведение к типу (то есть явная команда компилятору относиться к объекту как к одному из его присваиваемых типов) влияет на выбор перегруженных методов во время компиляции, но не на выбор переопределяемых методов.

На практике о том, что мы сейчас описали, не стоит часто беспокоиться, но это важно для понимания того, что делает и чего не делает виртуальная машина во время выполнения.

Связывание статических методов

Статические методы не принадлежат никаким экземплярам объектов; они вызываются непосредственно через имя класса, то есть не выбираются динамически во время выполнения, как методы экземпляра. Поэтому статические методы называются «статическими»; они всегда связываются во время компиляции.

Статический метод в надклассе может скрываться другим статическим методом в подклассе, пока исходный метод не объявлен финальным. Однако оба метода всегда вызываются напрямую, через соответствующие имена классов. Вы не можете «переопределить» статический метод методом экземпляра. Другими словами, вы не можете иметь статический метод и метод экземпляра с одной и той же подписью в той же иерархии класса.

Финальные методы и производительность

В таких языках, как C++, значение по умолчанию предназначено для методов, которые должны вести себя как скрытые переменные, поэтому вы должны явно объявлять методы, которые хотите видеть динамическими (или в терминах C++, *виртуальными*). В Java методы экземпляра по умолчанию динамические. Но вы можете использовать модификатор `final`, чтобы объявить, что метод экземпляра не может быть переопределен в подклассе и не будет субъектом динамического связывания.

Мы видели, что `final` используется с переменными, чтобы эффективно сделать их константами. Применимо к методу `final` означает, что его реализация является константой — переопределение не разрешено. `Final` также может применяться к целому классу, это означает, что в классе нельзя создавать подклассы.

Ранее связывание динамических методов вызывало снижение производительности, и некоторые до сих пор склоняются к использованию модификатора `final` для защиты от этого. Современные системы выполнения Java устраняют потребность в этом типе редактирования. Профилированное выполнение может определить, какие методы не переопределяются, и «оптимистично» встраивает их, относясь к ним, как к конечным, пока не станет необходимо поступить по-другому. Как правило, следует использовать или не использовать `final`, исходя из структуры вашей программы, а не из соображений ее выполнения.

Оптимизации компилятора

В некоторых более старых версиях Java компилятор `javac` может запускаться с опцией `-O`, которая сообщает ему, что некоторые оптимизации, такие как встраивание, следует выполнять статически. Большинство этих оптимизаций сейчас производится во время выполнения более умными виртуальными машинами, поэтому такие опции в основном не нужны.

Другой вид оптимизации позволяет вам включать код отладки в код Java, не опасаясь снижения производительности или увеличения размеров. Хотя в Java нет препроцессора для явного контроля того, что включается в источник, вы можете получить некоторые подобные эффекты, делая блок кода условным на константной переменной (то есть переменной `static` и `final`). Компилятор Java достаточно умен, чтобы убрать этот код, определив, что он не будет вызван. Например:

```
static final boolean DEBUG = false;
...
final void debug (String message) {
if (DEBUG) {
System.err.println(message);
// делать другие вещи
...
}
}
```

В этом случае компилятор может узнавать, что условие на переменной `DEBUG` всегда `false`, а тело метода `debug()` будет оптимизироваться. С современным компилятором вызов метода может даже полностью быть оптимизирован.

Отметьте, что этот вид отладки кода полезен для таких целей, как сбор данных. В отличие от рассмотренных в главе 4 утверждений, которые предположительно являются тестами на да/нет, гарантирующими верность вашей программной логики, эти условные блоки кода могут производить затратное форматирование или другую обработку выхода, которая полезна во время разработки, но которую вы не хотите иметь в конечном продукте.

Еще раз о выборе методов

Сейчас у вас должно быть хорошее интуитивное понимание того, как выбирать методы из совокупности потенциально перегруженных или переопределенных имен методов класса. Если, однако, вы жаждете подробностей, вы найдете их здесь.

В предыдущем разделе мы предложили индукционное правило для анализа перегруженных методов. В нем говорилось, что один метод считается более специфичным, чем другой, если его аргументы присваиваемы аргументам второго метода. Теперь мы можем расширить это правило, чтобы включить анализ переопределяемых методов, добавив следующее условие: один метод будет более точным, чем другой, если тип класса, содержащий метод, также назначается типу класса, хранящему второй метод.

Что это означает? Что ж, единственными классами, чьи типы являются назначаемыми, являются классы в той же иерархии наследования. Это означает, что мы говорим о наборе всех методов одного и того же имени в классе или в его родительских или дочерних классах. Поскольку типы подклассов назначаются типам надклассов, но не наоборот, анализ подталкивается путем, который мы ожидаем, по цепи подклассов. Это эффективно добавляет второе измерение в поиск, в котором анализ опускается по дереву наследования до более детализированных классов и, одновременно, до более специфично перегруженных методов внутри данного класса.

Исключения и переопределенные методы

Переопределенный метод может менять поведение объекта, но в некотором смысле он все еще должен выполнять договор оригинально-

го метода с пользователем. Так, переопределяемый метод должен придерживаться оператора `throws` в оригинальном методе. Новый метод не может выбрасывать новые типы проверяемых исключений. Он только может объявлять, что он выбрасывает типы исключений, назначаемые тем типам, которые выбрасываются методом в родительском классе; другими словами, он может объявить, что он выбрасывает те же типы исключений или подтипы тех исключений, объявленных исходным методом. Если новый метод не выбрасывает никаких проверяемых исключений оригинала, он не должен объявлять их, и операторы, вызывающие метод через подкласс, не должны предотвращать их. (Таким образом, вы можете перезаписать метод, чтобы «обработать» исключения для пользователя.)

Итак, новый метод может объявлять точно такие же проверяемые исключения, что и оригинал, или он имеет опцию для улучшения этих типов путем объявления того, что он выбрасывает более специфичные подтипы, чем переопределяемый метод. Это не равно утверждению, что метод может просто выбросить подтипы его объявленных исключений; любой метод может это сделать. Новый метод может фактически переопределить оператор метода `throws`, чтобы быть более специфичным. Эта техника называется *ковариантным* типом оператора `throws`, это означает, что типы исключения, от которых должен предохраняться пользователь, меняются, чтобы стать более улучшенными в подтипе.

Давайте быстро просмотрим, что на самом деле означает оператор `throws`. Если метод объявляет, что он может выбросить `IOException`, в действительности он говорит, что он может выбросить исключение типа `IOException` или его подтипов. Например, исключение `FileNotFoundException` является типом `IOException`. Метод, объявляющий, что он может выбросить `IOException`, может фактически выбросить `FileNotFoundException` или любой другой подтип `OException` во время выполнения:

```
public void readFile() throws IOException {
...
if ( error ) throw new FileNotFoundException( filename );
}
```

При вызове этого метода компилятор допускает возможность появления исключения любого типа, производного от `IOException`; при этом мы либо используем блок `try/catch`, либо генерируем исключение из нашего собственного метода.

Когда мы переопределяем метод в его подклассе, мы получаем возможность немного изменить оператор метода `throws`. Новый метод должен быть обратно совместимым с исходным, поэтому любое проверяемое исключение, которое он выбрасывает, должно быть назначаемым исключению, выбрасываемому переопределяемым методом. Но при желании мы можем быть более конкретными, улучшая тип исключения, чтобы поддерживать новое поведение метода. Например:

```
class MeatInedibleException extends InedibleException { ... }
class Animal {
void eat( Food f ) throws InedibleException {
...
}
class Herbivore extends Animal {
void eat( Food f ) throws MeatInedibleException {
if ( f instanceof Meat )
throw new MeatInedibleException();
...
}
}
```

В этом коде `Animal` определяет, что может сгенерировать исключение `InedibleException` в методе `eat()`. `Herbivore` является подклассом `Animal`, поэтому его метод `eat()` должен также быть способным выбросить `InedibleException`. Однако метод `eat()` класса `Herbivore` фактически объявляет, что выбрасывает более специфическое исключение: `MeatInedibleException`. Он может делать это, поскольку `MeatInedibleException` является подтипом `InedibleException`. Если мы работаем непосредственно с типом `Herbivore`, компилятор позволит нам отлавливать только `MeatInedibleException` и не потребует от нас защищаться от более общего `InedibleException`:

```
Herbivore creature = ...
try {
creature.eat( food );
} catch ( MeatInedibleException ) {
// creature не может есть эту пищу, поскольку это мясо
}
```

С другой стороны, если нас не волнует, почему еда несъедобна, нам ничто не мешает защищаться только от общего исключения `Inedible`

Exception и относиться к нему так же, как к любому другому классу Animal. Подведем итог: переопределяющий метод может улучшить не только поведение родительского метода, но и тип проверяемого исключения, которое он выбрасывает. Далее мы поговорим о перезаписанных методах, которые меняют свой возвращаемый тип точно таким же образом.

Возвращаемые типы и переопределенные методы

Чтобы метод дочернего класса считался переопределенным, он должен иметь точно такое же количество и типы аргументов. Он должен иметь такую же «информацию на входе», что и была. Как мы видели в предыдущем разделе, переопределяемые методы могут до какой-то степени улучшать свою «информацию на выходе». А именно они могут сузить их оператор throws, объявив, что они выбрасывают подтипы типов исключения оригинального метода. Что насчет главной «информации на выходе» метода? Его возвращаемого значения? Можем ли мы изменить возвращаемый тип метода, переопределяя его? Ответ заключается в том, что Java дает нам ковариантные возвращаемые типы для методов, так же, как она делает для типов исключений.

Это означает, что когда вы переопределяете метод, вы можете изменить возвращаемое значение на подтип возвращаемого значения оригинального метода. Например, если у нашего класса Animal есть генерирующий метод, называемый create(), который производит экземпляр Animal, наш класс Mammal может улучшить возвращаемый тип Mammal:

```
class Animal {
Animal create() { ... }
}
class Mammal extends Animal {
Mammal create() { ... }
}
```

Как мы увидим позже, эта методика очень полезна, поскольку в некоторых случаях она исключает приведение типов объектов во время выполнения программы.

Специальные ссылки: this и super

Специальные ссылки this и super позволяют вам ссылаться на члены экземпляра текущего объекта или члены надкласса соответственно. Мы видели, что this используется где угодно, чтобы передать ссылку

на текущий объект и сослаться на его скрытые переменные экземпляра. Ссылка `super` делает то же самое для родителей класса. Вы можете использовать ее, чтобы сослаться на члены надкласса, которые были скрыты или переопределены. То, что мы способны вызывать оригинальный метод надкласса, позволяет нам использовать его как часть нашего нового метода, делегируя его поведение до или после того, как мы выполнили дополнительную работу:

```
class Animal {
void eat( Food f ) throws InedibleException {
// употреблять еду
}
}
class Herbivore extends Animal {
void eat( Food f ) throws MeatInedibleException {
// проверить, является ли съедобным
...
try {
super.eat( f );
} catch ( InedibleException e ) { ... }
}
}
```

В этом примере наш класс `Herbivore` переопределяет метод `eat()` класса `Animal`, чтобы сначала выполнить проверку объекта `food`. После выполнения своей работы он использует `super.eat()`, чтобы вызвать реализацию `eat()` в его надклассе (в противном случае переопределенную или недоступную). `super` вызывает начало поиска метода или переменной в рамках ближайшего надкласса, а не в текущем классе. Найденный унаследованный метод или переменная могут принадлежать ближайшему надклассу или надклассу, находящемуся выше на дереве. Использование ссылки `super`, когда она применяется к перезаписанным методам надкласса, является особым случаем; он говорит системе разрешения метода завершить поиск динамического метода в надклассе вместо наиболее производного класса (как он делает в противном случае). Без `super` не будет никакого способа обратиться к переопределенным методам.

Приведение типов

Приведение явно говорит компилятору сменить видимый тип ссылки на объект. Приведение типов в основном употребляется, когда

объект временно приводится к более общему типу. Например, если бы строка была временно приведена к типу `Object` (с потенциальной возможностью использования ее в качестве объекта `String`), нам бы пришлось произвести приведение типов, чтобы снова получить исходный объект. Компилятор распознает только объявленные типы переменных и не знает, что мы в действительности поместили в него `String`. В Java приведения типов проверяются и во время компиляции, и во время выполнения, чтобы убедиться, что они правомерны. Во время компиляции компилятор Java остановит вас от попытки провести приведение типов, которое, возможно, не будет работать (такое, как превращение `Date` непосредственно в `String`). И во время выполнения Java будет удостоверяться, что те приведения типов, которые являются оправданными (такие, как из `Object` в `String`), в действительности корректны для воплощенных реальных объектов.

Попытка привести объект к несовместимому типу во время выполнения приводит к исключению `ClassCastException`. Только приведения между объектами, находящимися в той же иерархии наследования (и, как мы увидим позже, к подходящим интерфейсам) являются правомерными в Java и проходят проверку компилятора и системы выполнения. Приведения типов в Java влияют только на обращение со ссылками; они никогда не меняют форму действительного объекта. Это очень важное правило, которое нужно помнить. Вы никогда не меняете объект, к которому ведет ссылка, путем приведения его типа; вы меняете только понятие компилятора (или системы выполнения) о нем.

Приведение типов может использоваться для *сужения* или *сброса* типа ссылки — чтобы сделать ее более специфической. Часто мы будем делать это, когда нам будет нужно получить объект из более обобщенного типа коллекции или когда он до этого имел более общий тип. (Прототипическим примером является использование объекта в библиотеке, как мы увидим в главе 11.) Продолжим с нашим примером `Cat`:

```
Animal creature;
Cat simon;
// ...
creature = simon; // OK
// simon = creature; // Ошибка во время компиляции, несовместимый тип
simon = (Cat)creature; // OK
```

Мы не можем переназначить ссылку в `creature` на переменную `simon`, хоть мы и знаем, что она содержит экземпляр класса `Cat` (`Simon`).

Нам приходится производить указанное приведение типов, чтобы сузить ссылку. Обратите внимание, что косвенное приведение было произведено, когда мы перешли к другому способу расширения ссылки `simon` до типа `Animal` во время первого назначения. В этом случае ясное приведение было бы правомерным, но излишним.

Это все означает, что вы не можете соврать или догадаться о том, чем является объект. Если у вас есть объект `Cat`, вы можете использовать его как `Animal` или даже `Object`, поскольку все классы Java являются подклассами `Object`. Но если у вас есть `Object`, о котором вы думаете, что это `Cat`, вам придется выполнить приведение типа, чтобы он вернулся к `Animal` или `Cat`. Если вы не уверены, является ли класс `Object` объектом `Cat` или `Dog` во время выполнения, вы можете проверить это при помощи `instanceof` перед тем, как совершите приведение типов. Если вы не проверите и получите неверное приведение, система выполнения выдаст исключение `ClassCastException`.

```
if ( creature instanceof Cat ) {
Cat cat = (Cat)creature;
cat.meow();
}
```

Как мы упоминали ранее, приведение типов может повлиять на выбор элементов времени компиляции, таких как переменные и перегруженные методы, но не выбор переопределенных методов. Рис. 6.4 показывает разницу. Как показано в верхней половине диаграммы, приведение ссылки `simon` к типу `Animal` (ее расширение) влияет на выбор скрытой переменной `weight` внутри нее. Однако нижняя часть диаграммы указывает, что приведение не влияет на выбор переопределенного метода `sleep()`.

Приведение типов не панацея

Приведение типов — это то, чего стараются избежать программисты. Не только потому, что оно указывает на слабость в статической типизации кода, но и потому, что приведения типов могут быть просто громоздкими в использовании и делают код хуже читаемым. К сожалению, большей части кода, написанного на Java в прошлом, ничего не оставалось, кроме как положиться на приведение, чтобы он мог работать с любым типом объекта, в котором нуждается пользователь. Java 5.0 представил важное новое свойство языка, обобщение, отчасти для того чтобы решить эту проблему. Обобщение позволяет коду Java быть «на-

печатанным» для отдельного типа объекта от пользователя, уменьшая потребность в приведении типов во многих ситуациях. Мы детально обсудим обобщения в главе 8 и посмотрим, как они уменьшают потребность в приведении типов в большинстве кода Java.

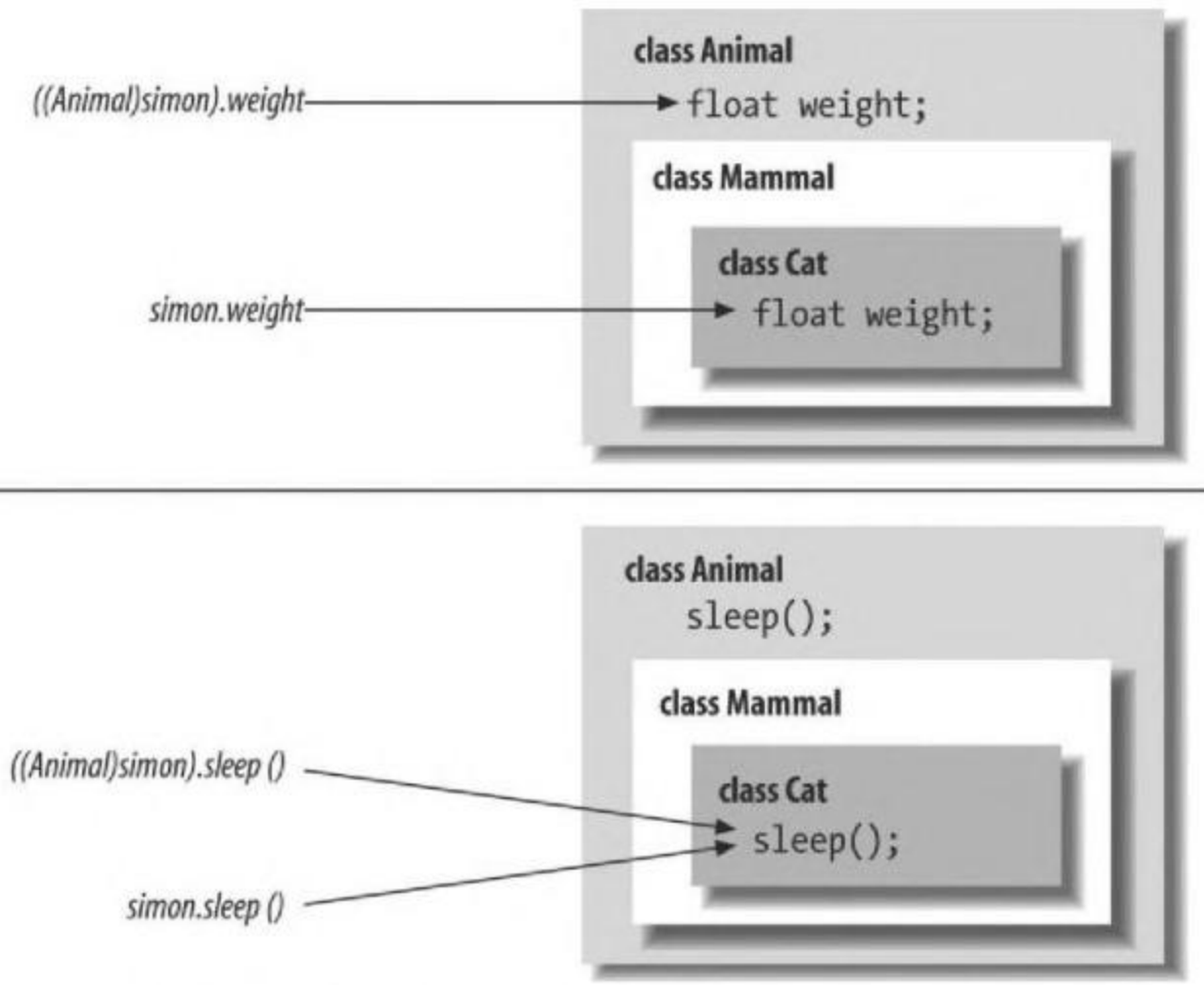

Рис. 6.4. Приведение и выбор методов и переменных

Использование конструкторов надклассов

Когда мы говорили ранее о конструкторах, мы обсуждали, как специальный оператор `this()` вызывает перегруженный конструктор по поступлении к другому конструктору. Подобным образом оператор `super()` явно вызывает конструктор надкласса. Конечно, мы также говорили о том, как Java создает цепь вызовов конструктора, которая включает конструктор надкласса, так почему не использовать `super()` явно? Когда Java осуществляет косвенный вызов конструктора надкласса, она вызывает конструктор по умолчанию. Если мы хотим вызвать конструктор надкласса, который принимает аргументы, мы должны это делать явно, используя `super()`.

Если мы собираемся вызвать конструктор надкласса при помощи метода `super()`, он должен находиться в самом начале нашего конструктора, так же как `this()` должен быть первым вызовом, который мы делаем в перегруженном конструкторе. Вот простой пример:

```
class Person {
Person ( String name ) {
// установка, основанная на имени
...
}
...
}
class Doctor extends Person {
Doctor ( String name, String specialty ) {
super( name );
// установка, основанная на специальности
...
}
...
}
```

В этом примере мы используем `super()`, чтобы получить преимущество реализации конструктора надкласса и избежать копирования кода для подготовки объекта, основанного на его имени. В действительности, поскольку класс `Person` не определяет конструктор по умолчанию (без аргументов), у нас нет другого выбора, кроме явного использования `super()`. В противном случае компилятор пожалуется, что он не смог найти подходящий конструктор по умолчанию для вызова. Другими словами, если вы унаследуете класс, все конструкторы которого принимают аргументы, вам придется вызвать явно один из конструкторов надкласса хотя бы из одного конструктора подкласса.

Переменные экземпляра класса инициализируются при возвращении от конструктора надкласса либо в связи с явным обращением к `super()`, либо с косвенным обращением к конструктору надкласса по умолчанию.

Полное разоблачение: конструкторы и инициализация

Теперь мы можем рассказать полную историю о том, как конструкторы соединяются друг с другом и когда возникает инициализация переменной. У правила есть три части, и оно применяется повторно для каждого последующего конструктора, который вызывается:

- если первый оператор конструктора является обычным оператором, то есть не является обращением к `this()` или `super()` — Java вставляет косвенное обращение к `super()`, чтобы вызвать конструктор су-

перкласса по умолчанию. После возвращения этого обращения Java инициализирует переменные экземпляра текущего класса и продолжает выполнять операторы текущего конструктора;

- если первый оператор конструктора является обращением к конструктору надкласса через `super()`, Java вызывает выбранный конструктор надкласса. После его возвращения Java инициализирует переменные текущего экземпляра класса и продолжает работать с операторами текущего конструктора;
- если первый оператор конструктора является обращением к перегруженному конструктору через `this()`, Java вызывает выбранный конструктор и после его возвращения просто продолжает обрабатывать операторы текущего конструктора. Обращение к конструктору надкласса произошло в пределах перегруженного конструктора либо явно, либо косвенно, поэтому инициализация переменных экземпляра уже произошла.

Абстрактные методы и классы

Чтобы сигнализировать о том, что метод является всего лишь прототипом, используется модификатор `abstract`. У абстрактного метода нет тела; это просто объявление сигнатуры, за которым следует точка с запятой. Вы не можете напрямую использовать класс, который содержит абстрактный метод; вместо этого вы должны создать подкласс, который реализует тело абстрактного метода:

```
abstract void vaporMethod( String name );
```

В Java класс, который содержит один или более абстрактных методов, должен быть явно объявлен как абстрактный класс, также при использовании модификатора `abstract`:

```
abstract classVaporClass {
...
abstract void vaporMethod( String name );
...
}
```

Абстрактный класс может содержать другие неабстрактные методы и обычное объявление переменной, но он не может быть проиллюстрирован. Чтобы его использовать, нужно создать подкласс от него и его абстрактные методы должны быть «переопределены» методами, кото-

рые реализуют тело. Не все абстрактные методы необходимо реализовывать в одном подклассе, но подкласс, который не переопределяет все абстрактные методы его надкласса фактическими, конкретными реализациями должен также быть объявлен при помощи `abstract`.

```
class MyVaporImplementation extends VaporClass {
void vaporMethod( String name ) { ... }
}
```

Абстрактные классы предоставляют каркас, который должен быть «заполнен» конструктором настоящих классов. Класс `java.io.InputStream`, например, имеет один абстрактный метод, который называется `read()`. Различные подклассы `InputStream` реализуют `read()` своими способами, чтобы читать со своих собственных источников. Остальная часть `InputStream`, однако, предоставляет расширенный функционал, построенный на простом методе `read()`. Подкласс `InputStream` наследует эти неабстрактные методы, чтобы предоставить функционал, основанный на простом методе `read()`, который реализует подкласс.

Интерфейсы

Java расширяет понятие абстрактного метода при помощи *интерфейсов*. Всегда желательно указать группу абстрактных методов, определяя некоторое поведение объекта, без привязки его к какой-либо реализации. В Java это называется интерфейсом. Интерфейс определяет набор методов, которые должен определять класс. Класс в Java может объявлять, что он *реализует* интерфейс, если он реализует необходимые методы. В отличие от расширения абстрактного класса, класс, реализующий интерфейс, не должен наследовать какую-то определенную часть иерархии наследования или использовать определенную реализацию.

Интерфейсы похожи на знаки отличия мальчиков-скаутов или девочек-скаутов. Скаут, который научился строить скворечник, может носить везде маленький шеврон с изображением скворечника. Это говорит миру: «Я знаю, как делать скворечники». Подобным образом интерфейс является списком методов, которые определяют некий набор поведений для объекта. Любой класс, который реализует каждый из перечисленных в интерфейсе методов, может объявить во время компиляции, что он реализует интерфейс и носит как свой знак отличия дополнительный тип — тип интерфейса.

Типы интерфейсов ведут себя как типы классов. Вы можете объявить, что переменные относятся к типу интерфейса, вы можете объявить аргументы методов, которые принимают типы интерфейса, и вы можете указать, что возвращаемый тип метода является типом интерфейса. В каждом случае подразумевается, что любой объект, который реализует интерфейс (то есть носит значок отличия), может выполнить эту роль. В этом смысле интерфейсы независимы от иерархии классов. Они идут в разрез с тем, каким типом объекта является элемент, и имеют дело только с ним и только в рамках того, что он может *делать*. Класс может применить столько интерфейсов, сколько он желает. Таким способом интерфейсы в Java возмещают потребность в множественном наследовании в других языках (вместе со всеми его неприятными трудностями).

По большому счету интерфейс выглядит полностью как класс `abstract` (то есть как класс только с методами `abstract`). Вы определяете интерфейс при помощи ключевого слова `interface` и списка методов без тел, только с прототипами (подписями):

```
interface Driveable {
boolean startEngine();
void stopEngine();
float accelerate( float acc );
boolean turn( Direction dir );
}
```

Предыдущий пример определяет интерфейс `Driveable`, который содержит четыре метода. Желательно, но не обязательно объявлять методы в интерфейсе при помощи модификатора `abstract`. Здесь мы этого не делали. Что более важно, методы интерфейса всегда считаются публичными и при желании вы можете объявлять их таковыми. Почему публичными? Что ж, в противном случае пользователь интерфейса не обязательно сможет его увидеть, и интерфейсы в основном предназначены для описания поведения объекта, а не его экземпляров.

Интерфейсы определяют возможности, поэтому их имена обычно указывают на то, для чего они предназначены. `Driveable`, `Runnable`, и `Updateable` являются хорошими названиями интерфейсов. Любой класс, который реализует все эти методы, может затем объявить, что он реализует интерфейс, используя специальный оператор `implements`, в его определении. Например:

```
class Automobile implements Driveable {
...
public boolean startEngine() {
if ( notTooCold )
engineRunning = true;
...
}
public void stopEngine() {
engineRunning = false;
}
public float accelerate( float acc ) {
...
}
public boolean turn( Direction dir ) {
...
}
...
}
```

Здесь класс `Automobile` реализует методы интерфейса `Driveable` и объявляет себя типом `Driveable`, используя ключевое слово `implements`.

Как показано на рис. 6.5, другой класс, такой как `Lawnmower`, может также реализовать интерфейс `Driveable`. Рисунок иллюстрирует то, что интерфейс `Driveable` применяется двумя разными классами. Хотя классы `Automobile` и `Lawnmower` могут иметь общий примитивный прототип, в нашем случае это всего лишь одна из возможностей.

После объявления интерфейса у нас был новый тип — `Driveable`. Мы можем объявить переменные типа `Driveable` и назначить им любой экземпляр объекта `Driveable`:

```
Automobile auto = new Automobile();
Lawnmower mower = new Lawnmower();
Driveable vehicle;
vehicle = auto;
vehicle.startEngine();
vehicle.stopEngine();
vehicle = mower;
vehicle.startEngine();
vehicle.stopEngine();
```

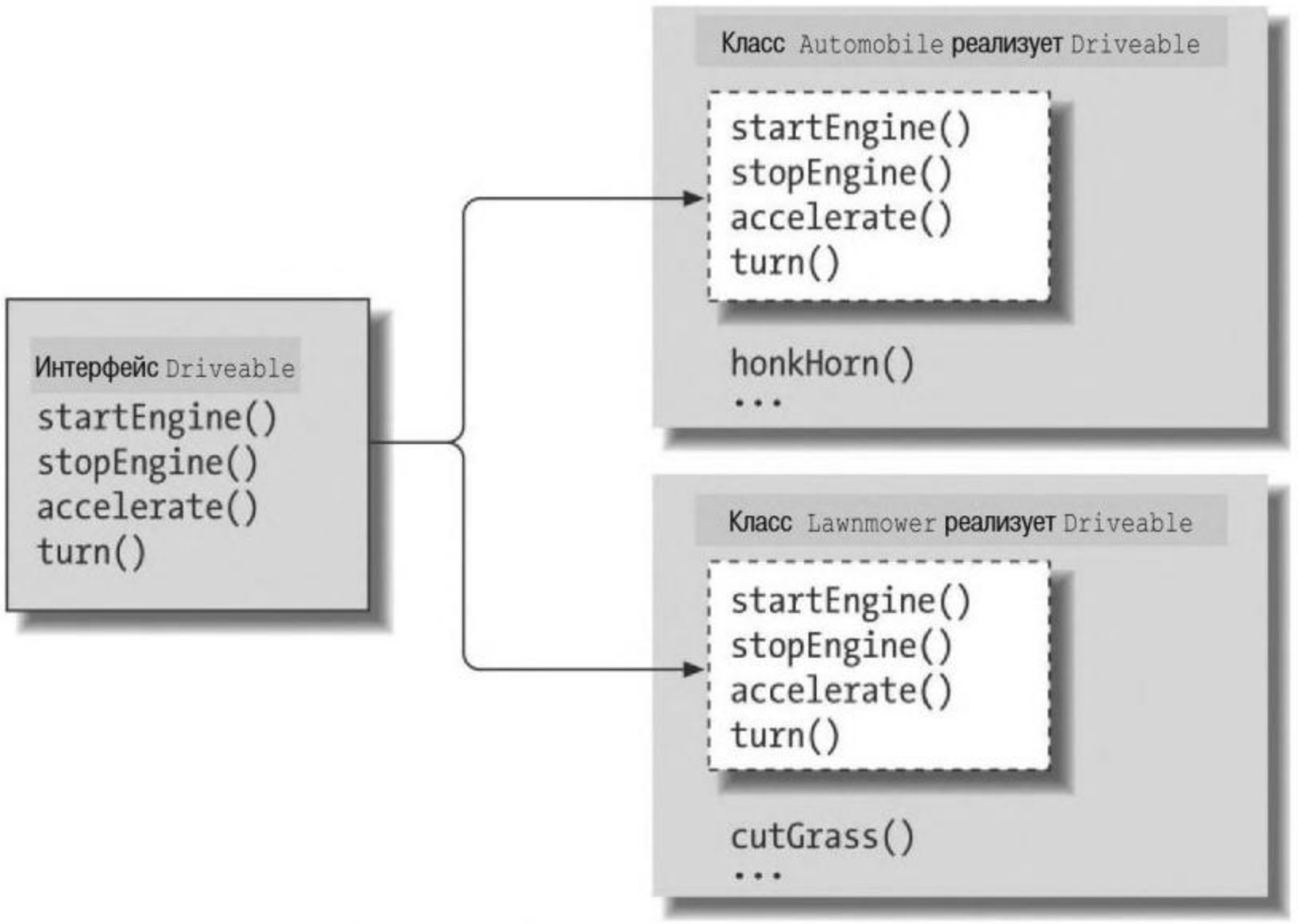

Рис. 6.5. Реализация интерфейса Driveable

`Automobile` и `Lawnmower` реализуют `Driveable`, поэтому они могут считаться взаимозаменяемыми объектами этого типа.

Интерфейсы и обратные вызовы

Интерфейсы могут использоваться для реализации «обратных вызовов» в Java. Это когда объект эффективно передает ссылку на один или более своих методов другому объекту. «Обратный вызов» возникает, когда вызываемый объект последовательно вызывает один из методов. В языке C или C++ это лучшая территория для указателей функции; Java вместо этого использует интерфейсы. В общем смысле эта концепция расширяется в Java до концепции *событий*, в которых объекты приемника регистрируются с источниками событий. Мы детально обсудим события в следующих главах.

Рассмотрите два класса: класс `TickerTape`, который показывает данные и класс `TextSource`, который предоставляет ленту информации. Мы бы хотели, чтобы наш `TextSource` отсылал любые новые текстовые данные. Мы бы могли заставить `TextSource` хранить ссылку на объект `TickerTape`, но тогда мы бы никогда не могли использовать наш

TextSource, чтобы отправлять данные любым другим типам объекта. Вместо этого нам бы пришлось размножить подклассы TextSource, которые работали с другими типами. Более простое решение — заставить TextSource хранить ссылку на тип интерфейса TextReceiver:

```
interface TextReceiver {
void receiveText( String text );
}
class TickerTape implements TextReceiver {
public void receiveText( String text ) {
System.out.println("TICKER:\n" + text + "\n");
}
}
class TextSource {
TextReceiver receiver;
TextSource( TextReceiver r ) {
receiver = r;
}
public void sendText( String s ) {
receiver.receiveText( s );
}
}
```

Единственное, о чем действительно заботится интерфейс TextSource, — нахождение правильного метода для вызова, чтобы вывести какой-то текст. Использование интерфейса устанавливает «контракт», receiveText(), для этого метода.

Когда TextSource создан, ссылка на TickerTape (который реализует интерфейс) хранится в переменной экземпляра. Это «регистрирует» TickerTape как «устройство вывода» TextSource. Всегда, когда ему нужно вывести данные, TextSource вызывает метод receiveText() устройства вывода. Далее мы увидим, что многие API в Java используют такую модель, но часто многие «приемники» могут регистрировать с теми же источниками.

Переменные интерфейса

Хотя интерфейсы в основном позволяют нам указывать поведение без реализации, есть одно исключение. Интерфейс может содержать константы (переменные static final), к которым можно обратиться

напрямую через имя интерфейса и которые также появляются в любом классе, реализуемом интерфейсом. Это позволяет константам упаковываться для использования с методами интерфейса:

```
interface Scaleable {
static final int BIG = 0, MEDIUM = 1, SMALL = 2;
void setScale( int size );
}
```

Интерфейс `Scaleable` определяет три целых числа `BIG`, `MEDIUM`, и `SMALL`. Все переменные, определенные в интерфейсах, косвенно являются переменными `final` и `static`; вам не нужно использовать модификаторы, но для ясности мы рекомендуем вам это сделать. Класс, который реализует `Scaleable`, видит эти константы:

```
class Box implements Scaleable {
void setScale( int size
switch( size ) {
case BIG:
...
case MEDIUM:
...
case SMALL:
...
}
}
...
}
```

Хотя технически с вашими интерфейсами в этом плане все правильно, смысл так делать исчез, когда Java добавила перечисления и *статические импорты*. Использовать интерфейсы для этой цели плохо, поскольку все эти публичные, статичные константы затем появляются в публичных API вашего класса и могут запутать тех, кто их использует. Что еще хуже, потом вы не сможете их удалить, поскольку другой код может полагаться на класс, который содержит эти значения.

Лучше использовать перечисление или поместить ваши константы в их собственный класс и затем использовать новый синтаксис статического импорта, чтобы устранить трудность обращения к ним. Мы обсудим статический импорт далее в этой главе. Этот отрывок кода даст вам представление о том, как это работает:

```
enum SizeConstants { BIG, MEDIUM, SMALL }
// использование
static import mypackage.SizeConstants;
...
setSize( MEDIUM );
```

Сигнальные интерфейсы

Иногда абсолютно пустые интерфейсы служат маркером того, что у класса есть специальное свойство. Интерфейс `java.io.Serializeable` является хорошим примером. Классы, которые реализуют интерфейс `Serializeable`, не должны добавлять никаких методов или переменных. Их дополнительный тип просто идентифицирует их в Java как классы, которые хотят быть способными к упорядочиванию. Это использование интерфейсов менее важно сейчас, когда в Java есть аннотации, описанные в главе 7.

Субинтерфейсы

Интерфейс может расширять другой интерфейс, также как класс может расширять другой класс. Такой интерфейс называется *субинтерфейсом*. Например:

```
interface DynamicallyScaleable extends Scaleable {
void changeScale( int size );
}
```

Интерфейс `DynamicallyScaleable` расширяет наш предыдущий интерфейс `Scaleable` и добавляет дополнительный метод. Класс, который реализует интерфейс `DynamicallyScaleable`, должен реализовать все методы обоих интерфейсов. Заметьте, что мы используем понятие *расширяет*, а не *реализует*, чтобы разделить интерфейс на подтипы. Интерфейсы не могут ничего реализовывать! Но интерфейс может расширять сколько угодно интерфейсов. Если вы хотите расширить два или более интерфейса, перечислите их после ключевого слова `extends`, разделив запятыми:

```
interface DynamicallyScaleable extends Scaleable, SomethingElseable {
...
}
```

Класс, который реализует этот интерфейс, также должен реализовывать другие интерфейсы. Более того, подтипы интерфейсов на-

значаются их надтипам таким же образом как классы, поэтому экземпляр `DynamicallyScaleable` может быть назначен переменной типа `Scaleable`.

Перекрывающие и конфликтующие методы

Мы также должны отметить вероятность того, что когда интерфейс расширяет два или более интерфейсов (или когда класс реализует два или более интерфейсов), в этих интерфейсах могут быть перекрывающие или конфликтующие методы. Если два метода в разных интерфейсах имеют одинаковую сигнатуру и возвращаемый тип, проблемы нет и реализация в классе удовлетворяет оба интерфейса. Если методы отличаются как перегруженные методы, класс должен реализовывать обе сигнатуры метода. Если у методов одинаковое имя, но они отличаются в возвращении типов исключений, класс не может реализовывать оба метода и возникают ошибки во время компиляции.

Пакеты и блоки компиляции

Пакетом называется группа родственных классов и интерфейсов. В главе 3 мы обсуждали, как Java использует имена пакетов, чтобы размещать классы во время компиляции и выполнения. В этом смысле пакеты похожи на библиотеки; они организуют наборы классов и управляют ими. Пакеты обеспечивают больше, чем просто организацию на уровне исходного кода. Они создают дополнительный уровень рамок для их классов и переменных и методов внутри них. Мы поговорим о видимости классов позже в этом разделе. В следующем разделе мы обсудим тот эффект, который пакеты оказывают на доступ к переменным и методам между классами.

Блоки компиляции

Исходный код классов Java организован в *блоки компиляции*. Простой блок компиляции содержит определение одного класса и называется, как этот класс. Определение класса `MyClass`, например, может возникать в файле, называющемся *MyClass.java*. Для большинства из нас блок компиляции — это просто файл с расширением .java, но теоретически в среде IDE он может быть произвольной сущностью. Для краткости мы обратимся к блоку компиляции просто как к файлу.

Разделение классов на их собственные файлы важно, поскольку компилятор Java предполагает большую ответственность утилиты `make` или `build`. Компилятор полагается на имена файлов источника, чтобы найти и скомпилировать зависимые классы.

Можно поместить более чем одно определение класса в один файл, но есть некоторые ограничения, которые мы вскоре обсудим.

Класс объявляется принадлежащим определенному пакету оператором `package`. Оператор `package` должен появиться как первый оператор в файле. Может быть только один оператор `package`, и он применяется ко всему файлу:

```
package mytools.text;
class TextComponent {
...
}
```

В этом примере класс `TextComponent` помещен в пакет `mytools.text`.

Имена пакетов

Имена пакетов по природе иерархические, в них используется способ наименования с разделением точками. По умолчанию компоненты имени пакета соответствуют именам директорий и служат уникальными путями для компилятора и системы выполнения для расположения источников файлов и классов Java. Однако, кроме расположения файлов, имена пакетов в Java не создают реальных отношений между пакетами. В действительности не существует такой вещи как «субпакет». Пространство имен пакета на самом деле горизонтальное, не иерархическое. Пакеты в отдельной части иерархии пакетов связаны только условно. Например, если мы создадим еще один пакет, называющийся `mytools.text.poetry` (предположительно для классов текста, которые в какой-то степени специализируются в работе с поэзией), эти классы не будут частью пакета `mytools.text`; они не будут иметь привилегий доступа членов пакета. В этом смысле способ наименования пакетов может ввести в заблуждение. Одно небольшое отклонение от этого принципа заключается в том, что утверждения, которые мы описали в главе 4, могут включаться и выключаться для пакета и пакетов «под» ним. Но это действительно просто дополнительное удобство, не представленное в структуре кода.

Видимость класса

По умолчанию класс доступен только другим классам в пакете. Это означает, что наш класс `TextComponent` доступен только для других классов в пакете `mytools.text`. Чтобы быть использованным вне пакета, класс должен объявляться как `public`:

```
package mytools.text;
public class TextEditor {
...
}
```

Теперь к классу `TextEditor` можно обратиться отовсюду. Файл источника кода Java может иметь только один класс `public`, определенный внутри него, и файл должен быть назван, как этот класс.

Пряча неважные и излишние классы, пакет создает *подсистему*, которая имеет четко определенный интерфейс для всего мира. Публичные классы предоставляют *фасад* для работы системы. Детали их внутренней работы могут оставаться скрытыми, как показано на рис. 6.6. В этом смысле пакеты могут прятать классы, как классы прячут приватные члены. Непубличные классы внутри пакета по этой причине иногда называются *package private*.

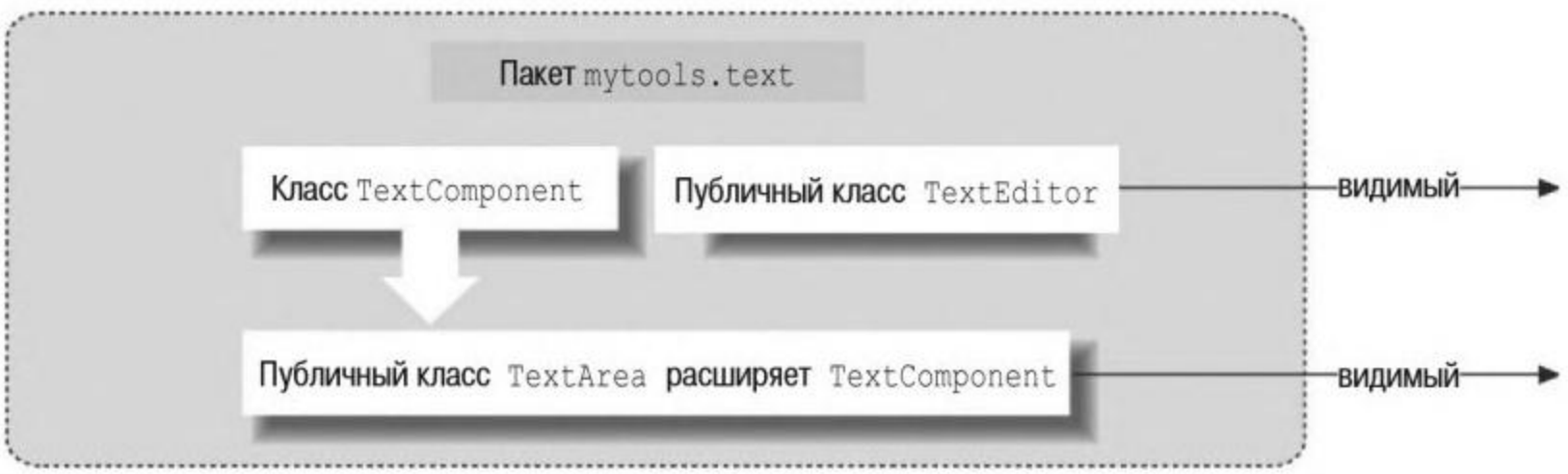

Рис. 6.6. Пакеты и видимость класса

Рис. 6.6 демонстрирует часть гипотетического пакета `mytools.text`. Классы `TextArea` и `TextEditor` объявлены как `public`, так что они могут использоваться в приложении везде. Класс `TextComponent` является частью реализации `TextArea` и не доступен вне пакета.

Импорт классов

Классы внутри пакета могут ссылаться друг на друга по своим простым именам. Однако, чтобы поместить класс в другой пакет, нам нужно

быть более конкретными. Продолжим предыдущий пример: приложение может прямо сослаться на наш класс редактора по его *полному имени* — `mytools.text.TextEditor`. Но мы быстро устанем от набора таких длинных имен классов, поэтому Java дает нам оператор `import`. Один или более операторов `import` могут появиться вверху компилируемого модуля после оператора `package`. Операторы `import` перечисляют полностью квалифицированные имена классов и пакетов, которые будут использоваться внутри файла.

Как оператор `package`, оператор `import` применяется ко всему компиляционному модулю. Вот как вы можете использовать оператор `import`:

```
package somewhere.else;
import mytools.text.TextEditor;
class MyClass {
TextEditor editBoy;
...
}
```

Как показано в этом примере, когда класс импортирован, на него можно ссылаться по его простому имени из любой части кода. Также возможно импортировать все классы в пакете, используя изображение символа обобщения:

```
import mytools.text.*;
```

Теперь мы можем ссылаться ко всем классам `public` в пакете `mytools.text` по их простым именам.

Очевидно, что может возникнуть проблема с импортом классов, которые имеют конфликтные имена. Компилятор не дает вам явно импортировать два класса с одним и тем же именем и выдает ошибку, если вы пытаетесь использовать неопределенный класс, который может прийти из двух пакетов, импортированных при помощи записи `import` пакета. В этом случае вам нужно только прибегнуть к полностью квалифицированным именам, чтобы обратиться к этим классам. Вы можете либо использовать полностью квалифицированное имя напрямую, либо добавить дополнительный оператор единственного класса `import`, который устраняет неоднозначность имени класса. Не имеет значения — до или после импорта пакета.

Кроме потенциальных конфликтов имен, у импорта большого количества классов нет минусов. Java не нагружает дополнительным бага-

жом файлы компилированных классов. Другими словами, файлы класса Java не содержат информацию об импортах; они только обращаются к классам, действительно используемым в них.



Одно замечание по поводу условных обозначений: в попытке оставить наши примеры короткими мы иногда будем импортировать целые пакеты (.*), даже когда станем использовать один класс или два из него. На практике обычно лучше быть точным, если возможно, и перечислять индивидуальные, полностью квалифицированные импорты классов, если их не много. Некоторые люди (особенно те, кто использует среду IDE, которая делает это за них) избегают использования импортов пакетов, предпочитая выводить каждый импортированный класс индивидуально. Как правило, лучшим выбором будет компромисс. Если вы собираетесь использовать более двух или трех классов из пакета, подумайте об импорте пакета.

Безымянный пакет

Класс, определенный в блоке компиляции, который не определяет пакет, входит в большой, неупорядоченный безымянный пакет. Классы в этом безымянном пакете могут ссылаться друг на друга по их простым именам. Их путь во время компиляции и во время выполнения считается текущей директорией, поэтому беспакетные классы полезны для экспериментов и тестирования (и для краткости в примерах книг о Java).

Операторы `import static`

`Import static` является разновидностью оператора `import`, который позволяет вам импортировать статические члены класса в пространство имен вашего файла, так что вам не нужно квалифицировать их, когда вы их используете. Лучшим примером этого является работа с классом `java.lang.Math`. Со статическим импортом мы можем достигнуть видимости встроенных математических «функций» и констант наподобие этого:

```
import static java.lang.Math.*;
// использование
double circumference = 2 * PI * radius;
double length = sin( theta ) * side;
int bigger = max( a, b );
int positive = abs( num );
```

Этот пример импортирует все статические члены класса `java.lang.Math`. Мы также можем импортировать индивидуальные члены по имени:

```
import static java.awt.Color.RED;
import static java.awt.Color.WHITE;
import static java.awt.Color.BLUE;
// использование
setField( BLUE );
setStripe( RED );
setStripe( WHITE );
```

Чтобы быть точными, эти статические импорты импортируют имя, а не специфический член в пространство имен нашего файла. Например, импорт имени foo импортирует все константы, названные `foo`, как любые методы, названные методом `foo()` в классе. Операторы `import static` наглядны и делают код более кратким. Однако их использование немного противоречит концепциям объектно-ориентированного программирования. Статические импорты являются лучшим решением для утилит и других методов общих возможностей, которые не требуют много контекста.

Видимость переменных и методов

Один из наиболее важных аспектов объектно-ориентированного программирования — это *сокрытие данных*, или *инкапсуляция*. Обращаясь с объектом в некоторых отношениях как с «черным ящиком» и игнорируя детали его реализации, мы можем писать более гибкий, простой код с компонентами, которые легко повторно использовать.

Базовые модификаторы доступа

По умолчанию переменные и методы класса доступны членам самого класса и других классов в том же пакете. Если использовать терминологию языка C++, классы в том же пакете являются *дружественными*. Мы назовем это дефолтным уровнем видимости. Как вы увидите по мере чтения, дефолтная видимость представляет собой средний из возможных уровней ограниченности.

С другой стороны, модификаторы `public` и `private` определяют крайние случаи. Как мы упомянули ранее, методы и переменные, объявленные как `private`, доступны только внутри их классов. В то же время члены, объявленные как `public`, доступны с любого класса в любом

пакете при условии того, что сам класс виден. (Класс, который содержит методы, должен также быть `public`, чтобы его можно было увидеть вне его пакета, как мы говорили ранее.) Члены класса, обозначенные как `public`, должны определять его самый общий функционал — то, что должен делать черный ящик.

Рис. 6.7 иллюстрирует четыре самых простых уровня видимости, продолжая пример из предыдущей главы. Публичные члены в `TextArea` доступны отовсюду. Приватные члены невидимы вне класса. Дефолтная видимость позволяет получить доступ к другим классам в пакете.

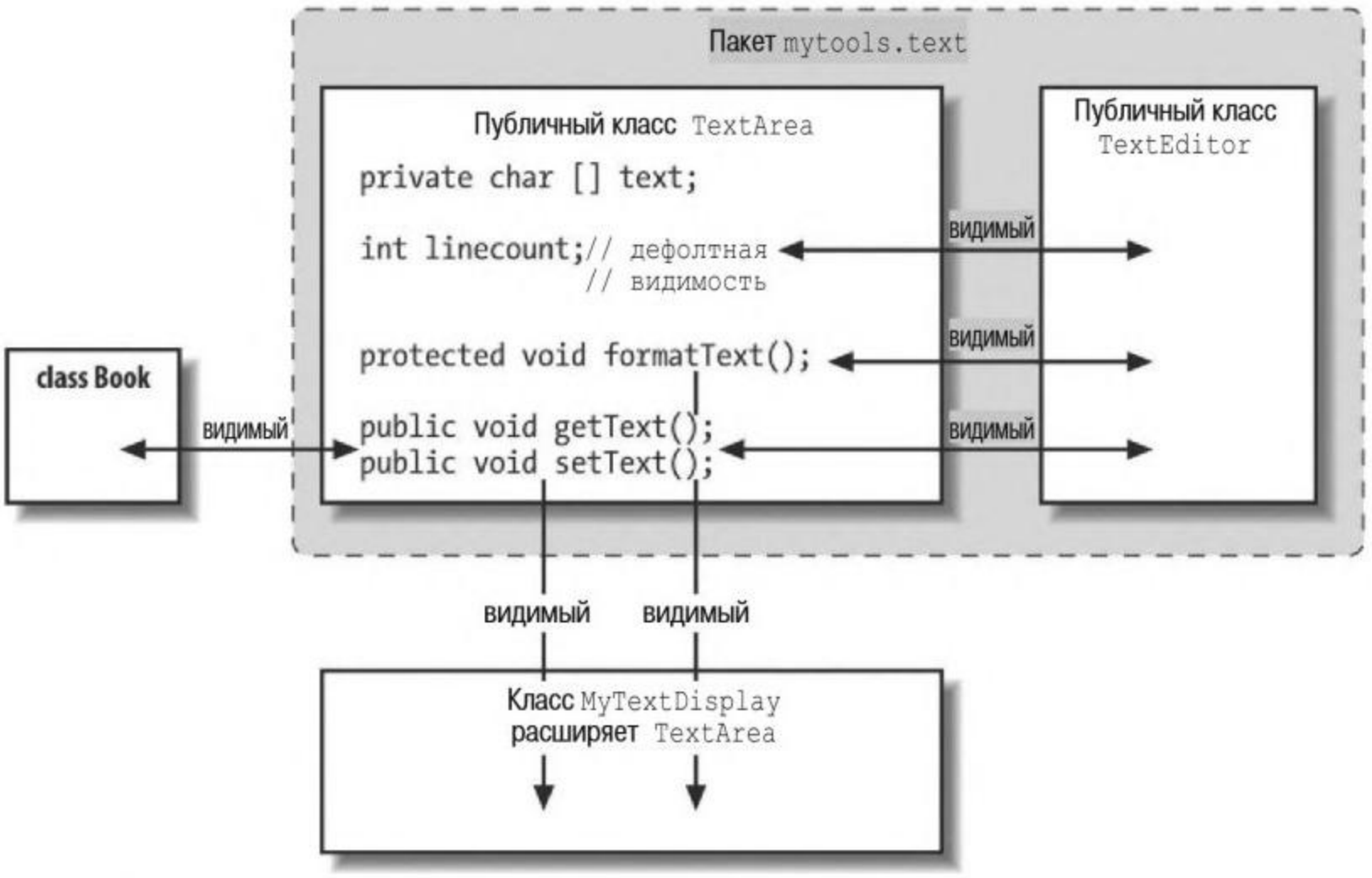

Рис. 6.7. Приватная, дефолтная, защищенная и публичная видимость.

Модификатор `protected` выдает специальные права доступа для подклассов. Как бы это ни прозвучало, `protected` лишь немного менее ограничителен, чем дефолтный уровень доступности. В дополнение к классам с разрешенным доступом по умолчанию в одном пакете члены `protected` видимы подклассам класса, даже если они определены в другом пакете. Если вы программист C++ и привыкли к более ограниченным значениям, это может направить вас по ложному пути*.

* На ранних стадиях язык Java допускал определенные комбинации модификаторов, одним из которых был `private protected`. Смыслом `private protected` было ограничивать видимость четко до подклассов (и устранить доступ к пакету). Позже это стало считаться запутанным и слишком сложным.

Табл. 6.1 кратко формулирует уровни видимости, доступные в Java; они располагаются от более до менее ограничивающих. Методы и переменные всегда видны внутри самого объявляющего класса, поэтому эти вопросы в таблице не отражены.

Табл. 6.1. Модификаторы видимости

Модификатор	**Видимость вне класса**
`private`	Нет
Модификатор отсутствует (по умолчанию)	Классы в пакете
`protected`	Классы в пакете и подклассы в пакете или вне его
`public`	Все классы

Подклассы и видимость

Подклассы привносят две важных (но не связанных) сложности в тему видимости. Во-первых, когда вы переопределяете методы в подклассе, переопределяющий метод должен иметь хотя бы такую же видимость, как переопределяемый метод. В то время как можно взять метод private и переопределить его методом `public` в подклассе, обратное невозможно; вы не можете переопределить метод `public` методом `private`. Это ограничение имеет смысл, если вы вспомните, что подтипы могут использоваться, как экземпляры их надтипа (например, `Mammal` является подклассом `Animal` и, таким образом, должен использоваться как `Animal`). Если бы мы могли переопределить метод менее видимым методом, у нас бы возникла проблема: есть вероятность, что наш `Mammal` не сможет делать то, что может `Animal`. Однако мы можем уменьшить видимость переменной. В этом случае переменная ведет себя как любая другая скрытая переменная; две переменные являются явными и могут иметь разную видимость в разных классах.

Следующую сложность немного тяжелее понять: переменные класса `protected` видимы его подклассам, но только через объекты типа подкласса или его подтипов. Другими словами, подкласс может видеть переменную `protected` своего надкласса как унаследованную переменную, но не может обратиться к этой же самой переменной через ссылку самого надкласса. Это утверждение может привести в замешательство, поскольку не вполне очевидно, что модификаторы видимости не ограничивают доступ между экземплярами того же класса таким же образом, каким они огра-

ничивают доступ между экземплярами разных классов. Два экземпляра одного класса могут обратиться ко всем методам друг друга, *включая приватные*, пока они обращаются друг к другу как к корректному типу. Другими словами, два экземпляра `Cat` могут обратиться ко всем переменным и методам друг друга (включая приватные), но `Cat` не может обратиться к защищенному члену в экземпляре `Animal`, если компилятор не сможет доказать, что `Animal` — это `Cat`. То есть классы `Cat` имеют привилегии того, что они являются `Animal` только по отношению к другим классам `Cat`, а не к любым классам `Animal`. Если вы находите это запутанным, не беспокойтесь. Если вы столкнулись с этой проблемой в реальности, вы, скорее всего, пытаетесь сделать что-то более сложное, чем следует.

Интерфейсы и видимость

Интерфейсы ведут себя как классы в пакетах. Интерфейс может быть объявлен public, чтобы быть видимым вне своего пакета. Под дефолтной видимостью интерфейс является видимым только внутри своего пакета. Как и в случае с классами, только один интерфейс public может быть объявлен в блоке компиляции (файле).

Массивы и иерархия классов

Сейчас мы немного притормозим и вернемся к теме массивов, рассматривая их с точки зрения объектов. В конце главы 4 мы упоминали, что массивы присутствуют в иерархии классов Java, но мы не вникали в этот вопрос. Теперь, когда мы обсудили объектно-ориентированные аспекты Java, мы можем показать вам всю картину.

Классы архивов живут в параллельной классу `Object` иерархии классов. Если класс является прямым подклассом `Object`, класс массива для этого базового типа также существует как базовый подкласс `Object`. Массивы более производных классов являются подклассами соответствующих классов массива. Например, рассмотрите следующие типы классов:

```
class Animal { ... }
class Bird extends Animal { ... }
class Penguin extends Bird { ... }
```

Рис. 6.8 иллюстрирует иерархию классов для массивов этих классов. Массивы того же размера связаны друг с другом таким же образом, как

их классы базового типа. В нашем примере Bird является подклассом Animal, что означает, что тип Bird[] является подтипом Animal[]. Таким же образом объект Bird может использоваться вместо объекта Animal, массив Bird[] может назначаться переменной типа Animal[]:

```
Animal [][] animals;
Bird [][] birds = new Bird [10][10];
birds[0][0] = new Bird();
// заставляет animals и birds ссылаться на один и тот же объект массива
animals = birds;
observe( animals[0][0] ); // обрабатывает объект Bird
```

Поскольку массивы являются частью иерархии классов, мы можем использовать instanceof, чтобы проверить тип массива:

```
if ( birds instanceof Animal[][] ) // истина
```

Массив является типом Object и, таким образом, может быть назначен переменным типа Object:

```
Object obj = animals;
```

Поскольку Java знает фактический тип всех объектов, вы также можете вернуться в исходное состояние, если это уместно:

```
animals = (Animal [][])something;
```

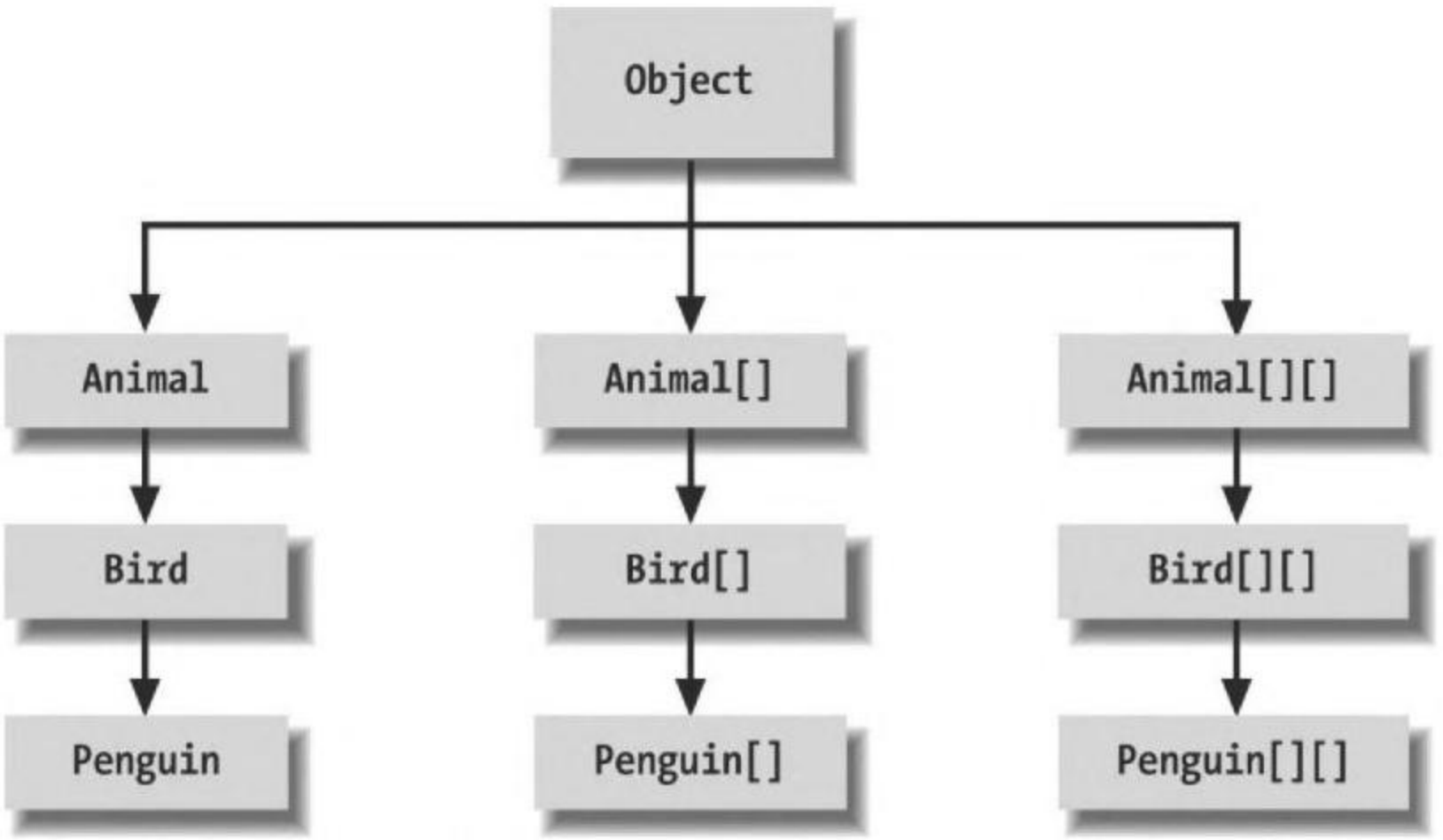

Рис. 6.8. Массивы в иерархии классов Java

Исключение `ArrayStoreException`

Поскольку у массивов есть свойство, что массив одного типа назначается массиву его надтипа, можно обманывать компилятор и пытаться запутать его, храня не тот тип объекта в массиве. Java может быть не в состоянии проверить типы всех объектов, которые вы помещаете в массивы во время компиляции. В этих случаях возможно получить исключение `ArrayStoreException` во время выполнения, если вы попытаетесь назначить неверный тип объекта массиву элемента. Например:

```
String [] strings = new String [10];
Object [] objects = strings; // выдает Object [] за String []
objects[0] = new Date(); // исключение ArrayStoreException во время
выполнения!
```

Здесь мы «присвоили псевдоним» `String []`, назначив его `Object []`. К третьей строке компилятор уже не знает настоящий тип массива, хранящегося в переменной объекта? и у него нет иного выбора, кроме как разрешить нам делать, что угодно. Конечно, во время выполнения виртуальная машина понимает, что мы пытается поместить объект `Date` в массив `String` и выбрасывает нам исключение `ArrayStoreException`. Эта проблема не должна встречаться вам часто при непосредственном использовании массива. Мы упоминаем ее здесь, поскольку мы снова встретимся с этим понятием, когда будем говорить об обобщениях в главе 8.

Внутренние классы

Все классы, которые мы видели в этой книге, были *высокоуровневыми*, «самостоятельными» классами, объявленными на уровне файла и пакета. Но на самом деле классы в Java могут быть объявлены на любом уровне границ, в любом наборе фигурных скобок (то есть практически везде, где вы можете поместить любое другое выражение Java). Эти *внутренние классы* принадлежат другому классу или методу, как принадлежала бы переменная, и могут иметь их видимость, ограниченную до ее границ таким же образом. Внутренние классы являются полезной и эстетически приятной возможностью для структуризации кода. Их кузены, *анонимные внутренние классы*, являются еще более мощным сокращением, с которым кажется, будто вы можете создавать новые виды объектов динамически внутри статически типизированной среды Java.

В Java анонимные внутренние классы играют роль *замкнутых* выражений в других языках, давая эффект управления состоянием и поведением отдельно от классов.

Однако по мере углубления в их внутреннюю работу, мы увидим, что внутренние классы не совсем так эстетически приятны или динамичны, как они кажутся. Внутренние классы являются чистым синтаксическим сахаром; они не поддерживаются виртуальной машиной и вместо этого обозначаются компилятором как обычные классы Java. Как программисту, вам возможно и не нужно этого знать; вы можете просто положиться на внутренние классы, как на любую другую языковую конструкцию. Однако вам нужно немного знать о том, как внутренние классы работают, чтобы лучше понимать компилируемый код и предвидеть несколько потенциальных побочных эффектов.

Внутренние классы являются фактически вложенными классами, например:

```
Class Animal {
Class Brain {
...
}
}
```

Здесь класс `Brain` является внутренним классом: это класс, объявленный внутри границ класса `Animal`. Хотя подробности того, что это означает, требуют некоторых пояснений, для начала скажем, что Java старается сделать значение насколько это возможно таким же, как и для других членов (методов и переменных), существующих на этом уровне границ. Например, давайте добавим метод классу `Animal`:

```
Class Animal {
Class Brain {
...
}
void performBehavior() { ... }
}
```

Класс `Brain` и метод `performBehavior()` находятся внутри границ `Animal`. Вследствие этого в любом месте внутри мы можем обращаться

Класс `Brain` и метод `performBehavior()` находятся внутри границ `Animal`. Вследствие этого в любом месте внутри мы можем обращаться к `Brain` и `performBehavior()` непосредственно по имени. Внутри `Animal` мы можем вызвать конструктор для `Brain` (`newBrain()`), чтобы получить объект `Brain` или вызвать `performBehavior()` для выполнения функции

того метода. Но ни Brain, ни performBehavior() обычно не доступны вне класса Animal без какой-то дополнительной квалификации.

В теле внутреннего класса Brain и теле метода performBehavior() мы имеем прямой доступ ко всем другим методам и переменным класса Animal. Итак, так же как метод performBehavior() мог работать с классом Brain и создавать экземпляры Brain, методы в классе Brain могут вызывать метод performBehavior() класса Animal и работать с любыми другими методами и переменными, объявленными в Animal. Класс Brain «видит» все методы и переменные класса Animal непосредственно в его границах.Последнее имеет важные последствия. Из класса Brain мы можем вызвать метод performBehavior(); то есть из экземпляра Brain мы можем вызвать метод performBehavior() экземпляра Animal. Что ж, какого экземпляра Animal? Если у нас есть несколько объектов Animal (скажем, несколько классов Cat и Dog), нам нужно узнать, чей метод performBehavior() мы вызываем. Что означает для определения класса быть «внутри» другого определения класса? Ответ состоит в том, что объект Brain всегда находится в одном экземпляре Animal: том, о котором ему сообщили, когда он был создан. Мы назовем объект, который содержит любой экземпляр Brain его *включающим экземпляром*.

Объект Brain не может находиться вне включающего экземпляра объекта Animal. Везде, где вы увидите экземпляр Brain, он будет привязан к экземпляру Animal. Хотя возможно создать объект Brain отовсюду (то есть из другого класса), Brain всегда требует включающего экземпляра Animal, чтобы «хранить» его. Мы теперь всегда будем говорить, что если к Brain нужно обратиться извне Animal, он будет вести себя наподобие класса Animal.Brain. Как с методом performBehavior(), модификаторы могут применяться для ограничения их видимости. Все обычные модификаторы видимости применяют, а внутренние классы могут также объявлять static, как мы увидим позже.

Мы сказали, что внутри класса Animal мы можем создать Brain обычным путем, используя, например, оператор new метода Brain(). Хотя нам вряд ли когда-нибудь это пригодится, мы также можем создать экземпляр Brain вне класса, сославшись на экземпляр Animal. Для этого должен быть доступен внутренний класс Brain и мы должны использовать специальную форму оператора new, разработанную как раз для внутренних классов:

```
Animal monkey = new Animal();
Animal.Brain monkeyBrain = monkey.new Brain();
```

Здесь экземпляр `monkey` класса `Animal` используется, чтобы квалифицировать оператор `new` для `Brain`. Опять же, это не очень распространенная вещь и вы можете просто забыть, что мы говорили об этом. Статические внутренние классы более полезны. Мы поговорим о них немного позже.

Внутренние классы и адаптеры

Особенно важно использовать внутренние классы для создания *классов адаптеров*. Класс адаптер является «вспомогательным» классом, который связывает один класс с другим очень специфическим путем. Используя классы адаптеров, вы можете писать свои классы более естественно, не предвосхищая каждую мыслимую потребность пользователя заранее. Вместо этого вы предоставляете классы адаптера, которые соединяют ваш класс с определенным интерфейсом. В качестве примера давайте скажем, что у нас есть объект `EmployeeList`:

```
public class EmployeeList {
private Employee [] employees = ... ;
...
}
```

Класс `EmployeeList` содержит информацию о ряде работников. Давайте скажем, что мы хотим, чтобы класс `EmployeeList` предоставлял свои элементы через итератор. Итератор является простым стандартным интерфейсом для последовательности объектов. У интерфейса `java.util.Iterator` есть несколько методов:

```
public interface Iterator {
boolean hasNext();
Object next();
void remove();
}
```

Он позволяет нам проходить через его элементы, спрашивая о следующем и проверяя, не осталось ли еще элементов. Итератор является хорошим кандидатом для класса адаптера, поскольку это интерфейс, который `EmployeeList` не может сам с готовностью реализовать. Почему список не может реализовать интерпретатор напрямую? Поскольку итератор — «односторонний», одноразовый взгляд на наши данные. Он не предназначен для того, чтобы перезапускаться и заново использоваться. Может быть необходимым, чтобы несколько итераторов прохо-

дило по списку в разных местах. Следовательно, мы должны держать реализацию итератора отдельно от самого класса `EmployeeList`. Это остро нуждается в простом классе для обеспечения производительности итератора. Но как должен выглядеть этот класс? До того, как мы узнали о внутренних классах, нам бы оставалось только создать новый класс «высшего уровня». Мы, наверное, были бы вынуждены вызвать ему `EmployeeListIterator`:

```
class EmployeeListIterator implements Iterator {
// множество знаний о EmployeeList
...
}
```

Здесь у нас есть комментарий, представляющий механизм, в котором нуждается класс `EmployeeListIterator`. Задумайтесь на секунду, что бы вам пришлось делать, чтобы реализовать этот механизм. Класс, получившийся в результате, был бы полностью связан с `EmployeeList` и не мог бы употребляться в других ситуациях. Что хуже, чтобы функционировать, он должен иметь доступ к внутренним механизмам `EmployeeList`. Нам придется позволить классу `EmployeeListIterator` обращаться к приватному массиву в `EmployeeList`, раскрывая эти данные более широко, чем нужно. Это далеко от идеала.

Это выглядит как подходящая работа для внутренних классов. Мы уже сказали, что `EmployeeListIterator` был бесполезен без `EmployeeList`; это во многом звучит как отношения «существование внутри», которые мы описывали ранее. Более того, внутренний класс позволяет нам избежать проблемы с инкапсуляцией, потому что он может обратиться ко всем членам его включающего экземпляра. Следовательно, если мы используем внутренний класс, чтобы реализовать итератор, массив `employees` может остаться `private`, невидимым вне `EmployeeList`. Поэтому давайте просто засунем этот вспомогательный класс внутрь границ нашего класса `EmployeeList`:

```
public class EmployeeList {
private Employee [] employees = ... ;
...
class Iterator implements java.util.Iterator {
int element = 0;
boolean hasNext() {
return element < employees.length ;
}
```

```
Object next() {
if ( hasNext() )
return employees[ element++ ];
else
throw new NoSuchElementException();
}
void remove() {
throw new UnsupportedOperationException();
}
}
}
```

Теперь класс `EmployeeList` может предоставить метод сродни следующему, чтобы позволить другим классам работать со списком:

```
Iterator getIterator() {
return new Iterator();
}
```

В результате мы можем проще относиться к именованию класса итератора. Поскольку это больше не высокоуровневый класс, мы можем дать ему имя, которое будет подходящим только внутри `EmployeeList`. В этом случае мы назвали его `Iterator`, чтобы подчеркнуть то, что он делает, но нам не нужно такого имени, как `EmployeeIterator`, которое показывает отношение к классу `EmployeeList`, поскольку оно косвенно. Мы также заполнили внутренность класса `Iterator`. Как вы можете видеть, теперь, когда он внутри рамок `EmployeeList`, `Iterator` имеет прямой доступ к своим приватным членам, поэтому он может напрямую обратиться к массиву `employees`. Это очень упрощает код и поддерживает безопасность во время компиляции.

Перед тем как продолжить, мы должны отметить, что внутренние классы могут иметь конструкторы, переменные и инициализаторы, в этом примере они нам были не нужны. Они во всех отношениях являются реальными классами.

Внутренние классы внутри методов

Внутренние классы могут также быть объявлены для «локального» использования внутри тела метода. Возвращаясь к классу `Animal`, мы можем поместить `Brain` внутрь метода `performBehavior()`, если мы решим, что класс полезен только внутри этого метода:

```
Class Animal {
void performBehavior() {
Class Brain {
...
}
}
}
```

В этой ситуации правила, регулирующие то, что может видеть `Brain`, те же, что и в нашем предыдущем примере. Тело `Brain` может видеть все, что находится в границах метода `performBehavior()` и вне их (в теле `Animal`). Это включает локальные переменные `performBehavior()` и его аргументы. Но из-за динамической природы вызова метода существует несколько ограничений и дополнительных оговорок, о чем мы расскажем в следующих главах. Если вы думаете, что внутренние классы внутри методов звучат заумно, подождите, пока мы доберемся до анонимных внутренних классов, которые чрезвычайно полезны.

Ограничения, связанные с классами внутри методов

`performBehavior()` является методом, а вызов метода имеет ограниченное время действия. Когда они завершают выполнения, их локальные переменные обычно исчезают в никуда. Однако экземпляр `Brain` (как и любые объекты, созданные внутри метода) живет, пока к нему обращаются. Язык Java должен проверять, что любые используемые экземплярами `Brain` локальные переменные, созданные при вызове `performBehavior()`, также продолжают существовать. Более того, все экземпляры `Brain`, которые мы создаем в одном вызове `performBehavior()`, должны видеть те же локальные переменные. Чтобы сделать это, компилятор должен иметь разрешение на создание копий локальных переменных. Таким образом, их значения не могут меняться после того, как внутренний класс их видел. Это означает, что любые локальные переменные метода или аргументы, к которым обращается внутренний класс, должны быть объявлены как `final`. Модификатор `final` означает, что они константны после назначения. Это немного тяжело запомнить, но компилятор любезно напомнит вам. Например:

```
void performBehavior( final boolean nocturnal )
{
class Brain {
void sleep() {
```

```
if ( nocturnal ) { ... }
}
}
}
```

В этом отрывке кода аргумент `nocturnal` метода `performBehavior()` должен быть отмечен как `final`, так что к нему можно ссылаться внутри внутреннего класса `Brain`. Это просто техническое ограничение того, как применяются внутренние классы, гарантирующее то, что для класса `Brain` хранить копию значения, это норма.

Внутренние статические классы

Ранее мы упоминали, что внутренний класс `Brain` класса `Animal` может в каком-то смысле считаться классом `Animal`, просто используя такое квалифицированное имя: `Animal.Brain`. Но, как мы описали, при условии того, что наш класс `Animal.Brain` всегда требует экземпляра `Animal` как его включающего экземпляра, работать непосредственно этим способом довольно нетипично.

Однако есть другая ситуация, в которой мы хотим использовать внутренние классы по имени. Внутренний класс, который существует внутри тела высокоуровневого класса (не внутри метода или другого внутреннего класса), может быть объявлен как `static`. Например:

```
class Animal {
static class MigrationPattern {
...
}
```

Такой статический внутренний класс ведет себя как новый высокоуровневый класс, называющийся `Animal.MigrationPattern`. Мы можем использовать его, как любой другой класс, не придавая значения никаким включающим экземплярам. Хотя это может показаться странным, противоречия тут нет, поскольку статический член никогда не имеет объекта, связанного с ним. Требование, что внутренний класс должен быть объявлен непосредственно внутри высокоуровневого класса, обеспечивает то, что включающий экземпляр не будет необходим. Если у нас есть разрешение, мы можем создать экземпляр класса, используя квалифицированное имя:

```
Animal.MigrationPattern stlToSanFrancisco =
new Animal.MigrationPattern();
```

Как вы видите, результатом является то, что `Animal` ведет себя как мини-пакет, храня класс `MigrationPattern`. Здесь мы использовали полностью квалифицированное имя, но мы также могли импортировать его как любой другой класс:

```
import Animal.MigrationPattern;
```

Этот оператор разрешает нам обращаться к классу просто как к `MigrationPattern`. Мы можем использовать все стандартные модификаторы видимости для внутренних классов, поэтому статический внутренний класс может иметь видимость `private`, `protected`, `default` или `public`.

Вот еще один пример. Интерфейс Java 2D API использует статические внутренние классы, чтобы реализовать специализированные классы форм (то есть класс `java.awt.geom.Rectangle2D` имеет два внутренних класса, `Float` и `Double`, которые реализуют две разные величины). Эти классы форм в действительности являются очень простыми подклассами; было бы грустно умножать количество высокоуровневых классов в этом пакете на три, чтобы приспособить их все. При помощи внутренних классов мы можем объединить их с их соответствующими классами:

```
Rectangle2D.Float rect = new Rectangle2D.Float();
```

Анонимные внутренние классы

Теперь мы добрались до лучшей части. Согласно общему правилу, чем более инкапсулированы и ограничены в масштабах наши классы, тем большую свободу мы имеем в их наименовании. Мы видели это ранее в нашем примере с итератором. Это не только вопрос эстетики. Обычно мы хотим использовать самые краткие осмысленные имена, какие только возможно. Следствием из этого является то, что мы стараемся не давать имена полностью эфемерным объектам, которые будут использоваться только один раз.

Анонимные внутренние классы являются расширением синтаксиса операции `new`. Когда вы создаете анонимный внутренний класс, вы объединяете объявление класса с локализацией экземпляра этого класса, эффективно создавая «разовый» класс и экземпляр класса в одном

операторе. После ключевого слова new вы указываете либо имя класса, либо интерфейс, за которым следует тело класса. Тело класса становится внутренним классом, который либо расширяет указанный класс, либо, в случае интерфейса, предполагаемо реализует интерфейс. Единственный экземпляр класса создается и возвращается как значение.

Например, мы можем избавиться от объявления класса Iterator в примере EmployeeList, используя анонимный внутренний класс в методе getIterator():

```
Iterator getIterator()
{
return new Iterator() {
int element = 0;
boolean hasNext() {
return element < employees.length ;
}
Object next() {
if ( hasNext() )
return employees[ element++ ];
else
throw new NoSuchElementException();
}
void remove() {
throw new UnsupportedOperationException();
}
};
}
```

Здесь мы просто переместили внутреннее содержимое класса Iterator в тело анонимного внутреннего класса. Обращение к new косвенно создает класс, который реализует интерфейс Iterator и возвращает экземпляр класса в качестве результата. Отметьте размер фигурных скобок и точку с запятой в конце. Метод getIterator() содержит единственный оператор — return.

Предыдущий пример — это скорее крайний случай, и уж точно не улучшает читабельность. Внутренние классы лучше всего использовать, когда вы хотите реализовать несколько строк кода, но многословие и очевидность объявления отдельного класса отвлекают от имеющегося задания. Вот пример лучше. Предположим, что мы хотим запустить новый поток для выполнения метода performBehavior() нашего Animal:

```
new Thread() {
public void run() { performBehavior(); }
}.start();
```

Здесь мы проявили лаконичность. Мы локализовали и начали новый `Thread`, используя анонимный внутренний класс, который расширяет класс `Thread` и вызывает наш метод `performBehavior()` в своем методе `run()`. Эффект аналогичен использованию указателя метода в каком-нибудь другом языке. Однако внутренний класс позволяет компилятору проверить последовательность методов, что было бы более сложно (или невозможно) с настоящим указателем метода. В то же время наш класс анонимного адаптера с тремя строчками кода является намного более эффективным и читабельным, чем создание нового высокоуровневого класса адаптера, называющегося `AnimalBehaviorThreadAdapter`.

Хотя мы немного бежим впереди паровоза, анонимные адаптеры класса являются идеально совместимыми с обработкой событий (которую мы рассмотрим в полном объеме в главе 16). Избегая большого количества объяснений, давайте предположим, что вы хотите, чтобы метод `handleClicks()` вызывался всегда, когда пользователь щелкает мышью. Вы бы написали вот такой код:

```
addMouseListener( new MouseInputAdapter() {
public void mouseClicked(MouseEvent e) { handleClicks(e); }
} );
```

В этом случае анонимный класс расширит класс `MouseInputAdapter`, переопределив метод `mouseClicked()` для вызова нашего метода. Многое происходит в очень малом пространстве, но результатом является чистый, читабельный код. Вы назначаете имена методов, которые имеют смысл для вас, позволяя Java выполнять свою работу по проверке типов.

Определение области видимости ссылки this

Иногда внутренний класс может не справиться с включающим экземпляром его «родителя». Он, возможно, захочет передать ссылку своему родителю или сослаться на одну из переменных родителя или метода, которая была скрыта своей переменной. Например:

```
class Animal {
int size;
class Brain {
```

```
int size;
}
}
```

Здесь, пока затрагивается `Brain`, переменная `size` в `Animal` скрывается ее собственной версией.

Обычно объект ссылается сам на себя, используя специальную ссылку `this` (косвенно или явно). Но каков же смысл `this` для объекта с одним или несколькими включающими экземплярами? Ответ заключается в том, что внутренний класс имеет множество ссылок `this`. Вы можете уточнить, какую ссылку `this` вы хотите, добавив префикс с именем класса. Например, мы можем получить ссылку к нашему классу `Animal` из `Brain` таким образом:

```
class Brain {
Animal ourAnimal = Animal.this;
...
}
```

Подобным образом мы можем обратиться к переменной `size` в классе `Animal`:

```
class Brain {
int animalSize = Animal.this.size;
...
}
```

Как в действительности работают внутренние классы?

В конце концов, давайте засучим рукава и посмотрим, что в действительности происходит, когда мы используем внутренний класс. Мы сказали, что компилятор делает все, о чем мы надеялись забыть. Давайте посмотрим, что в действительности происходит. Попробуйте скомпилировать этот простой пример:

```
class Animal {
class Brain {
}
}
```

Вы обнаружите, что компилятор сгенерирует два файла *.class: Animal.class* и *Animal$Brain.class*.

Второй файл является файлом класса для нашего внутреннего класса. Да, как мы и боялись, внутренние классы — это и правда магия компилятора. Компилятор создал для нас внутренний класс как обычный высокоуровневый класс и назвал его, объединив знак доллара с именами классов. Знак доллара является валидным символом в именах классов, но предназначен для использования только автоматизированными элементами. (Пожалуйста, не начинайте называть ваши классы знаком доллара.) Если бы наш класс был глубже расположен, промежуточные имена внутренних классов присоединялись бы таким же образом для создания уникальных высокоуровневых имен.

Теперь посмотрите на класс при помощи утилиты javap в JDK. Начиная с Java 5.0, вы можете сослаться к внутреннему классу как `Animal.Brain`, но в более ранних версиях Java, вам, возможно, пришлось бы вызывать класс по его настоящему имени — `Animal$Brain`:

```
% javap 'Animal$Brain'
class Animal$Brain extends java.lang.Object {
Animal$Brain(Animal);
}
```

В системе Windows нет необходимости заключать в кавычки аргумент, как мы делали это в командной строке Unix.

Вы увидите, что компилятор дал нашему внутреннему классу конструктор, который принимает ссылку на `Animal` в качестве аргумента. Так реальный внутренний класс получает ссылку на его включающий экземпляр.

Худшее в этих дополнительных файлах класса — вам нужно знать, что они там. Такие утилиты, как `jar`, автоматически их не находят; когда вы вызываете такую утилиту, вам нужно указать эти файлы явно или использовать знак обобщения, чтобы найти их:

```
% jar cvf animal.jar Animal*class
```

Замечания относительно безопасности

Принимая во внимание то, что мы только что видели, — что внутренний класс действительно существует как автоматически сгенерированный высокоуровневый класс, как он получает доступ к приватным переменным? Ответ, к сожалению, состоит в том, что компилятор вынужден прерывать инкапсуляцию вашего объекта и вставлять метод доступа,

так что внутренний класс может получить к ним доступ. Методам доступа обеспечен доступ на уровне пакета, поэтому ваш объект все еще находится в безопасности внутри стен пакета, но значимость этой разности обманчива, если людям было разрешено создавать новые классы в вашем пакете.

С модификаторами видимости во внутренних классах также связаны некоторые проблемы. Текущие реализации виртуальной машины не реализуют понятие класса `private` или `protected` в пакете, поэтому предоставление вашему классу другой видимости, кроме `public` или дефолтной, является только гарантией времени компиляции. Тяжело представить, как эту безопасность можно нарушить, но интересно отметить, что Java немного напрягается, чтобы придерживаться своего исходного замысла*.

Глава 7
РАБОТА С ОБЪЕКТАМИ И КЛАССАМИ

В предыдущих двух главах мы узнали объекты Java и их взаимоотношения. Теперь мы взберемся по ступенькам иерархии классов Java до самого верха и закончим изучение базового языка в наивысшей точке. В этой главе мы поговорим о самом классе `Object`, который является «бабушкой» всех классов Java. Мы также опишем более фундаментальный класс `Class` (класс, называющийся Class), который представляет классы Java в виртуальной машине Java. Мы обсудим, что мы можем делать с этими компонентами независимо друг от друга. Это приведет нас к более общей теме: API отражения Java, которое позволяет программам Java инспектировать объекты (возможно, неизвестные) и работать с ними динамически во время выполнения. В конце концов мы поговорим об API аннотаций Java, которое позволяет разработчикам добавлять метаданные в источник кода для использования компилятором и системами выполнения, которые их ищут.

Класс `Object`

`java.lang.Object` является прародителем всех объектов; это изначальный класс, из которого в конечном счете произведены все остальные классы. Методы, определенные в `Object`, следовательно, очень важны, поскольку они появляются в каждом экземпляре каждого класса, во всей Java. По последним подсчетам было девять публичных методов и два защищенных метода в классе `Object`. Пять из них являются версиями `wait()` и `notify()`, которые используются для синхронизации потоков в экземплярах объекта, как мы обсудим в главе 9. Остальные четыре метода используются для базового сравнения, трансформирования и управления.

Каждый объект имеет метод `toString()`, который может быть вызван, когда он должен быть представлен как текстовое значение. Объекты `PrintStream` используют метод `toString()`, чтобы выводить данные, как описано в главе 12. `toString()` также используется косвенно, ког-

да на объект ссылаются в конкатенации строк. Ниже представлено несколько примеров:

```
MyObj myObject = new MyObj();
Answer theAnswer = new Answer();
System.out.println( myObject );
String s = "The answer is: " + theAnswer ;
```

Новый тип объекта может переопределить `toString()` и реализовать собственную версию, которая предоставляет необходимую информацию о себе. Это особенно полезно при отладке, когда часто выводится значение строки объекта, чтобы видеть, что происходит. Два других метода, `equals()` и `hashCode()`, могут также требовать конкретизации, когда вы создаете новый класс.

Равенство и эквивалентность

Метод `equals()` определяет, являются ли два объекта эквивалентными. Что именно это означает для каждого класса — решать вам. Два объекта `String`, например, считаются эквивалентными, если они хранят точно такие же символы в той же последовательности:

```
String userName = "Joe";
...
if ( userName.equals( suspectName ) )
arrest( userName );
```

Использование метода `equals()` — это не то же самое, что оператор == в Java:

```
if ( userName == suspectName ) // Неверно!
```

Этот оператор проверяет, относятся ли две переменные ссылок, `userName` и `suspectName`, к одному и тому же объекту. Это проверка на *идентичность*, а не на равенство. Две переменные, которые являются идентичными (указывают на один и тот же объект), обязательно будут равны, но обратное не всегда правда. В Java возможно создать два объекта `String` с одинаковым контентом, которые тем не менее являются разными экземплярами класса `String` — хотя, как мы обсудим позже, Java пытается помочь нам избежать этого.

Класс должен переопределить метод `equals()`, если ему необходимо реализовать собственное обозначение равенства. Если у вас нет необ-

ходимости сравнивать объекты отдельного класса, вам не обязательно переопределять `equals()`.

Обращайте особое внимание на случайную перегрузку метода `equals()`, если вы хотите переопределить его. При перегрузке сигнатуры метода отличаются; при переопределении они должны оставаться такими же. Сигнатура метода `equals()` указывает на аргумент `Object`, так что объект может быть сравнен с любым другим видом объекта, не только с объектом своего типа класса. Вы, возможно, захотите проверять на эквивалентность только объекты того же типа. Но чтобы переопределить (не перегрузить) `equals()`, метод должен указать, что его объект должен быть представителем класса `Object`.

Вот пример того, как правильно переопределять метод `equals()` в классе `Shoes` методом `equals()` в подклассе `Sneakers`. Используя собственный метод, объект `Sneakers` может сравнить себя с любым другим объектом:

```
class Sneakers extends Shoes {
public boolean equals( Object arg ) {
if ( (arg != null) && (arg instanceof Sneakers) ) {
// сравнить arg с этим объектом, чтобы проверить равенство
// если сравнение справедливо…
return true;
}
return false;
}
...
}
```

Если бы мы указали `public boolean equals(Sneakers arg) ...` в классе `Sneakers` мы бы перегрузили метод `equals()` вместо переопределения. Если случится, что другой объект будет назначен переменной не из класса `Sneakers`, сигнатура метода не совпадет. Результат: вызывается реализация метода `equals()` надкласса `Shoes`, что может быть, а может и не быть тем, чего вы ожидали.

Хеш-коды

Метод `hashCode()` возвращает целое число, которое является хеш-кодом объекта. *Хеш-код* — что-то наподобие сигнатуры или контрольной суммы объекта; это случайный идентифицирующий номер, который в действи-

тельности генерируется из контента объекта. Хеш-код всегда должен отличаться от экземпляров класса, которые содержат разные данные, но должен быть таким же для экземпляров, которые сравнивают «равное» с методом `equals()`. Хеш-коды используются в процессе хранения объектов в библиотеке `Hashtable` или подобных библиотеках. (Библиотека `Hashtable` иногда называется словарем или ассоциативным массивом в других языках.) случайное распространение значений хеш-кода помогает `Hashtable` оптимизировать хранение объектов, служа идентификатором при распределении их для хранения и затем ровно и быстро располагая их.

Стандартная реализация `hashCode()` в `Object` в действительности не реализует эту схему. Вместо этого она назначает каждому экземпляру объекта уникальный номер. Если вы не переопределяете этот метод, когда создаете подкласс, каждый экземпляр вашего класса имеет уникальный хеш-код. Этого достаточно для некоторых объектов. Однако, если ваши классы имеют эквивалентные объекты (если вы переопределили `equals()`) и вы хотите, чтобы равные объекты служили эквивалентными ключами в `Hashtable`, следует переопределить `hashCode()`, так что ваши эквивалентные объекты будут генерировать те же самые значения хеш-кода. Мы вернемся к теме хеш-кодов в главе 11, когда будем обсуждать классы `Hashtable` и `HashMap`.

Клонирование объектов

Объекты могут использовать метод `clone()` класса `Object`, чтобы создавать свои копии. Скопированный объект является новым экземпляром объекта, отдельным от оригинала. Он может содержать, а может и не содержать точно такое же состояние (такие же значения экземпляра), что и оригинал; это контролируется объектом, который копируется. Также важно: решение о том, позволяет ли вообще объект клонировать себя, зависит от объекта.

Класс `Object` в Java предоставляет механизм для создания простой копии объекта, включая все его «пустые» состояния — *побитовую* копию. Но по умолчанию эта возможность выключена. (Вскоре мы покажем почему.) Чтобы сделать себя клонируемым, объект должен реализовать интерфейс `java.lang.Cloneable`. Это интерфейс-метка, который указывает Java, что объект хочет сотрудничать в том, чтобы быть клонированным (интерфейс в действительности не содержит никаких методов). Если объект не является клонируемым, метод `clone()` выбрасывает исключение `CloneNotSupportedException`.

Метод `clone()` является защищенным, поэтому по умолчанию он может быть вызван только объектом на себя, объектом в том же пакете или другим объектом такого же типа или подтипа. Если мы хотим, чтобы кто угодно мог клонировать наш объект, мы должны переопределить метод `clone()` и сделать его публичным.

Ниже представлен простой клонируемый класс — `Sheep`:

```
import java.util.HashMap;
public class Sheep implements Cloneable {
HashMap flock = new HashMap();
public Object clone() {
try {
return super.clone();
} catch (CloneNotSupportedException e ) {
throw new Error(
"Это никогда не должно происходить, потому что мы реализовали интерфейс
Cloneable!");
}
}
}
```

`Sheep` имеет одну переменную экземпляра, `HashMap`, называемую `flock` (которую овца использует, чтобы отслеживать свою подругу овцу). Наш класс реализует интерфейс `Cloneable`, показывая, что можно копировать `Sheep` и что он переопределил метод клон, чтобы сделать его публичным. Наш метод `clone()` просто возвращает объект, созданный методом `clone()` надкласса — копию нашей `Sheep`. К сожалению, компилятор не настолько умен, чтобы догадаться, что объект, который мы клонируем, никогда не выбросит исключение `CloneNotSupportedException`, поэтому мы в любом случае должны принять меры предосторожности. Теперь наша овца является клонируемой. Мы можем создавать такие копии:

```
Sheep one = new Sheep();
Sheep anotherOne = (Sheep)one.clone();
```

Приведение типов здесь необходимо, поскольку возвращаемым типом метода `clone()` является `Object`. Мы можем сделать лучше, изменив возвращаемый тип переопределенного метода `clone()` в подклассе и передвинув приведение типов в сам метод `clone()`, чтобы сделать все немного проще на стороне пользователей класса:

```
public Sheep clone() {
try {
return (Sheep)super.clone();
} catch (CloneNotSupportedException e ) {
throw new Error("Этого никогда не должно случиться!");
}
}
// использование
Sheep one = new Sheep();
Sheep anotherOne = one.clone();
```

В любом случае теперь у нас есть две овцы вместо одной. Правильно примененный метод `equals()` скажет нам, что овцы эквивалентны, но `==` говорит нам, что они в действительности являются двумя различными экземплярами класса `Sheep`.

Java создал *пустую копию* для нашего класса `Sheep`. Что же в ней такого пустого? Язык Java просто скопировал значения наших переменных. Это означает, что переменная экземпляра `flock` в каждом нашем классе `Sheep` до сих пор хранит ту же самую информацию, то есть обе овцы имеют ссылку на один и тот же класс `HashMap`. Ситуация выглядит, как показано на рис. 7.1.

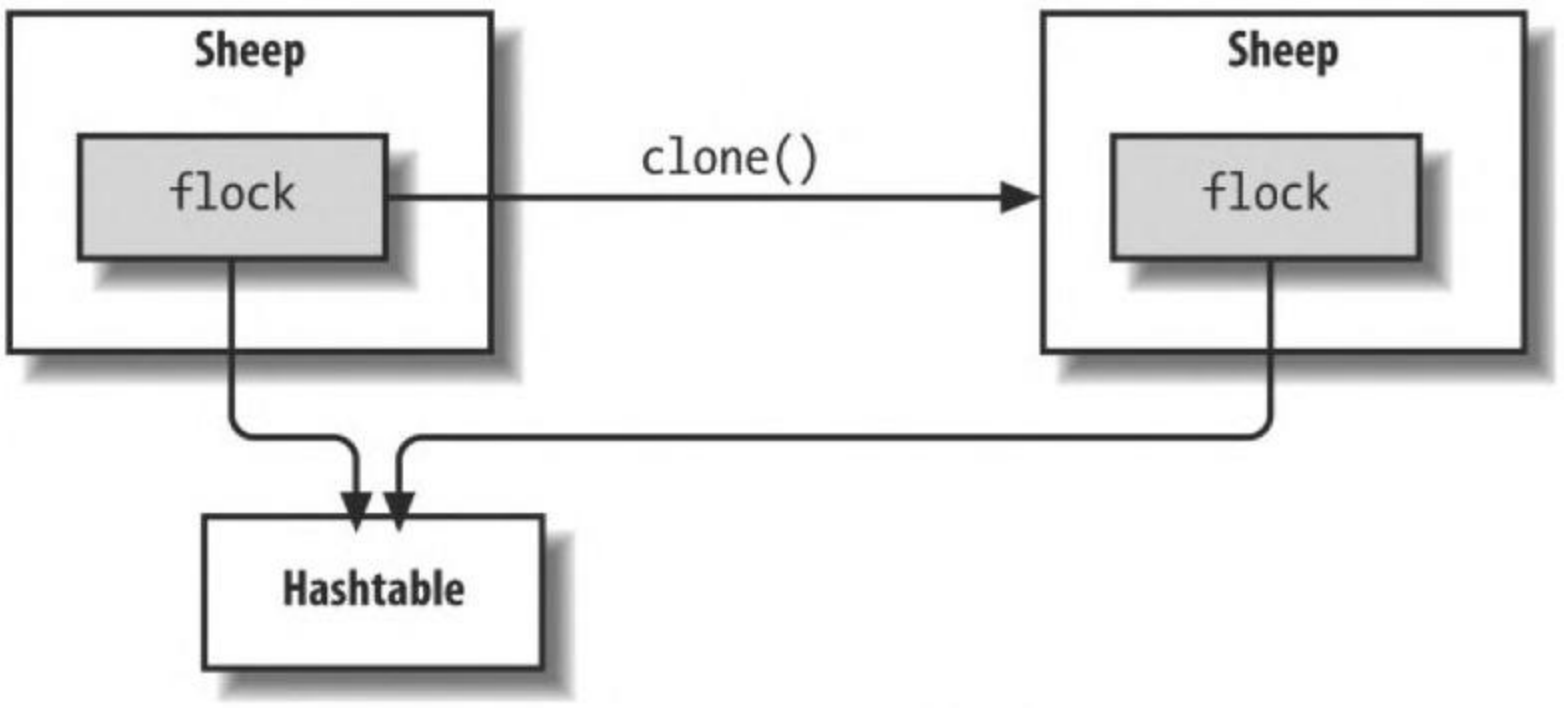

Рис. 7.1. Пустая копия объекта

Это может быть, а может и не быть тем, чего вы хотели. Если вместо этого мы пожелаем, чтобы наш класс `Sheep` имел отдельные копии ее полного состояния (или чего-то среднего), мы можем взять управление в свои руки. В следующем примере, `DeepSheep`, мы реализуем *детальную копию*, дублирующую наш собственный класс `flock`:

```
public class DeepSheep implements Cloneable {
HashMap flock = new HashMap();
public DeepSheep clone() {
try {
DeepSheep copy = (DeepSheep)super.clone();
copy.flock = (HashMap)flock.clone();
return copy;
} catch (CloneNotSupportedException e ) {
throw new Error("Этого никогда не должно случиться!");
}
}
}
```

Наш метод `clone()` теперь также клонирует класс `HashMap`. Теперь, когда класс `DeepSheep` клонирован, ситуация выглядит больше похожей на рис. 7.2.

Каждый класс `DeepSheep` теперь имеет собственную полную копию карты, которая не может содержать разные элементы. Теперь вы можете видеть, почему объекты не являются клонируемыми по умолчанию. Не имело бы никакого смысла предполагать, что все объекты могут быть рационально продублированы с пустой копией. Подобным образом нет смысла предполагать, что глубокая копия необходима или даже верна. В этом случае нам нет необходимости делать глубокую копию; класс `flock` содержит те же члены вне зависимости от того, на какой класс `sheep` вы смотрите, поэтому нет смысла копировать `HashMap`. Но решение зависит от самого объекта и его требований.

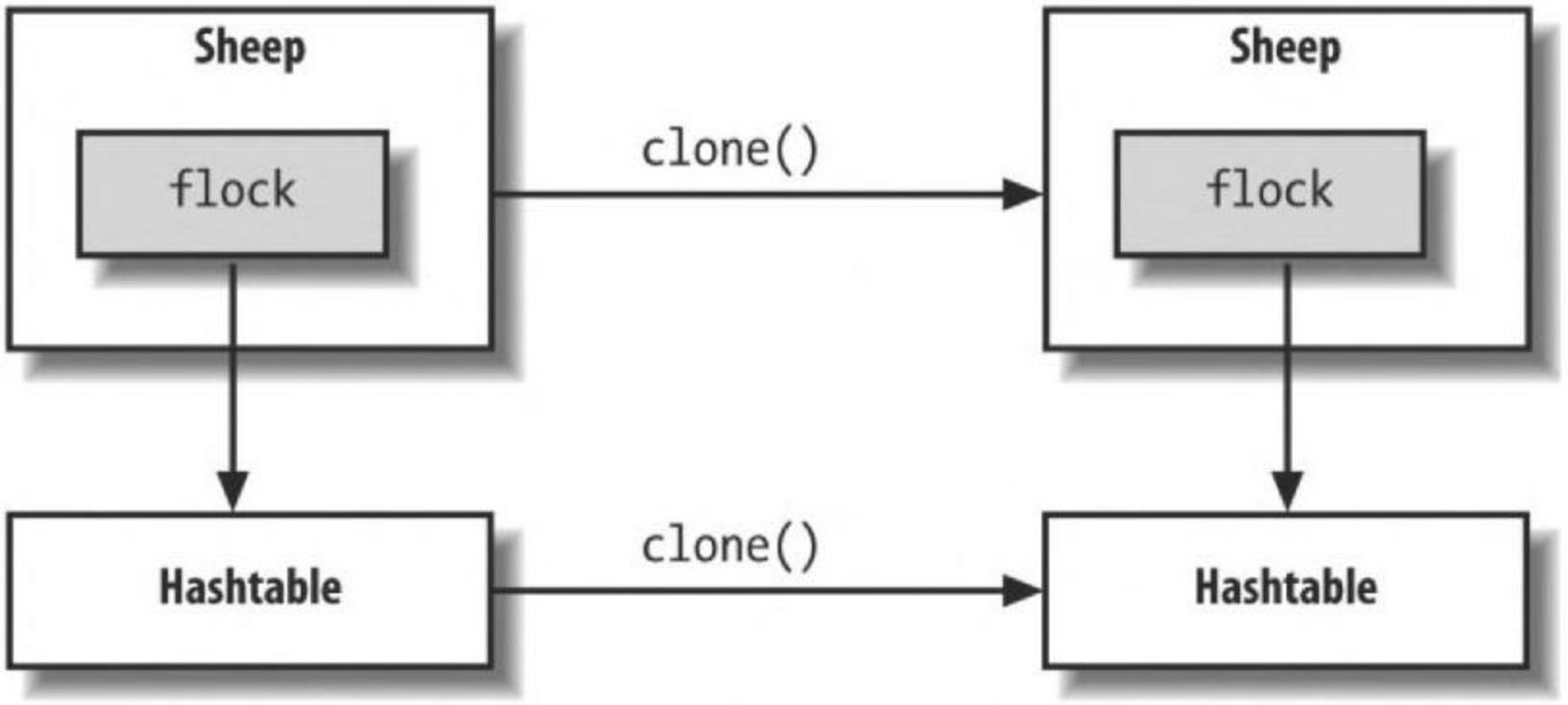

Рис. 7.2. Глубокая копия объекта

Последний метод класса `Object`, который мы должны обсудить, — это метод `getClass()`. Этот метод возвращает ссылку на объект `Class`, который создал экземпляр `Object`. Мы поговорим об этом далее.

Класс Class

Хорошей мерой сложности в объектно-ориентированном программировании является степень абстрагирования от его структур класса. Мы знаем, что каждый объект в Java является экземпляром класса, но что же в действительности такое класс? В таких языках, как традиционный язык C++, объекты формулируются и подвергаются обработке классов, но классы на самом деле являются артефактами компилятора. В этих языках вы видите, что классы упоминаются только в исходном коде, не во время выполнения. Для сравнения: классы в Smalltalk являются настоящими сущностями выполнения в языке, которые сами описаны как «метаклассы» или «метаклассы классов». Java нашла золотую середину между двумя этими языками в двухзвенной системе, которая использует объекты класса `Class`.

Классы в исходном коде Java представлены во время выполнения экземплярами класса `java.lang.Class`. Существует объект `Class` для каждого типа объектов, который вы используете; этот объект `Class` отвечает за создание экземпляров этого типа. Но в основном вам не нужно заботиться об этом, если вы не заинтересованы в загрузке новых типов классов динамически во время выполнения или использовании очень абстрагированной API, которая хочет «тип» вместо фактического аргумента. Объект `Class` также является базой для «отражения» класса, чтобы найти его методы и другие свойства, позволяя вам выяснить структуру объекта или вызвать его методы программным путем во время выполнения. Мы обсудим отражения в следующей главе.

Мы ассоциируем `Class` с отдельным объектом при помощи метода `getClass()`:

```
String myString = "Foo!"
Class stringClass = myString.getClass();
```

Мы можем также сделать, чтобы `Class` ссылался на отдельный класс статически, используя запись `.class`:

```
Class stringClass = String.class;
```

Ссылка `.class` выглядит как статическое поле, которое существует в каждом классе. Однако в действительности решение принимает компилятор.

Что мы можем сделать с объектом `Class` — это спросить его полное имя:

```
String s = "Boofa!";
Class stringClass = s.getClass();
System.out.println( stringClass.getName() ); // "java.lang.String"
```

Еще одна вещь, которую мы можем сделать с `Class`, — это попросить его создать новый экземпляр его типа объекта. Продолжая с предыдущим примером:

```
try {
String s2 = (String)stringClass.newInstance();
}
catch ( InstantiationException e ) { ... }
catch ( IllegalAccessException e ) { ... }
```

Здесь `newInstance()` имеет возвращаемый тип класса `Object`, поэтому мы должны привести его к ссылке подходящего типа. Это хорошо, но в следующей главе мы увидим, что класс `Class` является обобщенным классом, и это означает, что мы можем параметризировать его, чтобы он был более специфичным по отношению к типу Java, с которым мы имеем дело; то есть мы можем получить метод `newInstance()`, чтобы вернуть правильный тип непосредственно без приведения. Продемонстрируем это, но не беспокойтесь, если пока не видите смысла:

```
Class<String> stringClass = String.class;
try {
String s2 = stringClass.newInstance(); // нет необходимости в приведении типов
}
catch ( InstantiationException e ) { ... }
catch ( IllegalAccessException e ) { ... }
```

Здесь может быть выброшена пара исключений. `InstantiationException` указывает, что мы пытаемся создать экземпляр класса `abstract` или интерфейса. `IllegalAccessException` является более общим исключением, которое указывает, что мы не можем обратиться к конструктору за объектом. Отметьте, что `newInstance()` может создать только экземпляр класса, который имеет доступный дефолтный конструктор. Он

не позволяет нам передать никакие аргументы конструктору. (В следующем разделе мы узнаем как сделать это, используя API отражения.)

Все это становится более осмысленным, когда мы добавляем способность искать класс по имени. `forName()` — это статический метод класса `Class`, который возвращает объект, названный объектом класса `String`:

```
try {
Class sneakersClass = Class.forName("Sneakers");
} catch ( ClassNotFoundException e ) { ... }
```

Исключение `ClassNotFoundException` выбрасывается, если класс не может быть обнаружен.

Комбинируя эти инструменты, мы получаем мощную способность загружать новые типы классов динамически. При соединении с мощностью интерфейсов мы можем использовать новые типы данных, загруженные именем строки в наших приложениях:

```
interface Typewriter {
void typeLine( String s );
...
}
class Printer implements Typewriter {
...
}
class MyApplication {
...
String outputDeviceName = "Printer";
try {
Class newClass = Class.forName( outputDeviceName );
Typewriter device = (Typewriter)newClass.newInstance();
...
device.typeLine("Hello...");
}
catch ( Exception e ) { ... }
}
```

Здесь приложение загружает реализацию класса (`Printer`, который реализует интерфейс `Typewriter`), зная только его имя. Представьте, что имя было введено пользователем или найдено в файле конфигурации. Этот тип загрузки класса является базой для многих типов настраиваемых систем в Java.

Отражение

В этой главе мы рассмотрим API отражения Java, поддерживаемые классами в пакете `java.lang.reflect`. Как следует из названия, отражение — это способность класса или объекта исследовать себя. Отражение позволяет коду Java смотреть на объект (если быть более точными, класс объекта) и определять его структуру. В пределах, заданных диспетчером безопасности, вы можете узнать, какие конструкторы, методы и поля имеет класс, а также его атрибуты. Вы даже можете менять значение полей, динамически вызывая методы и конструировать новые объекты, как если бы Java имела простые указатели переменных и методов. И вы можете делать все это с объектами, которые ваш код никогда не видел ранее. API аннотаций также имеет способность сохранять метаданные об исходном коде в компилированных классах, и мы можем получить эту информацию при помощи API отражения. Нам не хватит места в этой книге, чтобы в полной степени рассмотреть API отражения. Пакет `reflect` сложен и богат деталями. Но отражение разработано так, что вы можете делать многое, не затрачивая много усилий; 20% усилий дают вам 80% веселья.

API отражения может использоваться для определения способностей объектов во время выполнения. Она используется сериализацией объектов для передачи в потоках или для постоянного хранения. Очевидно, способность расщеплять объекты и видеть их содержимое должна рьяно охраняться диспетчером безопасности. Общим правилом является то, что вашему коду не позволено делать с API отражения ничего, чего он не может сделать со статическим (обычным, компилированным) кодом Java. Вкратце: Java является мощным инструментом, но не автоматической лазейкой. По умолчанию объект не может использовать работу с полями или методами, к которой обычно он не может обратиться (например, приватные поля другого метода), хотя эти привилегии могут быть выданы, как мы обсудим позже.

Три первичных свойства класса — это его поля (переменные), методы и конструкторы. Для целей описания и обращения к объекту эти три свойства представлены отдельными классами в API отражения: `java.lang.reflect.Field`, `java.lang.reflect.Method` и `java.lang.reflect.Constructor`. Мы можем искать эти члены класса через объект класса `Class`.

Класс `Class` предоставляет две пары методов для достижения каждого типа свойства. Одна пара позволяет обратиться к публичным свойствам класса (включая унаследованные от надклассов), в то время как

другая пара позволяет обратиться к любому публичному или непубличному элементу, объявленному непосредственно в классе (но не к свойствам, которые унаследованы), с соблюдением соображений безопасности. Некоторые примеры:

- `getFields()` возвращает массив объектов класса `Field`, представляющих все публичные переменные класса, включая те, которые он наследует.
- `getDeclaredFields()` возвращает массив, представляющий все переменные, объявленные в классе, независимо от их модификаторов доступа, но не включающих унаследованные переменные.
- Для конструкторов различие между «все конструкторы» и «объявленные конструкторы» является незначительным (классы не наследуют конструкторы), поэтому `getConstructors()` и `getDeclaredConstructors()` отличаются только в том, что первый возвращает публичные конструкторы, в то время как последний возвращает все конструкторы класса.

Каждый набор методов включает методы для приема всех элементов сразу (например, `getFields()`) и дополнительный метод для поиска отдельного элемента по имени и — по методу и конструктору — по сигнатуре (например, `getField()`, который принимает имя поля в качестве аргумента).

Следующий список показывает методы в классе Class:

```
Field [] getFields();
```

Получает все публичные переменные, включая унаследованные.

```
Field getField(String name);
```

Получает указанные публичные переменные, которые могут быть унаследованы.

```
Field [] getDeclaredFields();
```

Получает все публичные и непубличные переменные, объявленные в этом классе (не включая те, которые унаследованы от надкласса).

```
Field getDeclaredField(String name);
```

Получает указанную переменную, публичную или непубличную, объявленную в этом классе (унаследованные переменные не считаются).

```
Method [] getMethods();
```

Получает все публичные методы, включая унаследованные.

```
Method getMethod(String name, Class ... argumentTypes);
```

Получает указанный публичный метод, который имеет аргументы, совпадающие с типами, перечисленными в `argumentTypes`. Метод может быть унаследованным.

```
Method [] getDeclaredMethods();
```

Получает все публичные и непубличные методы, объявленные в этом классе (не включая те, которые были унаследованы от надклассов).

```
Method getDeclaredMethod(String name, Class ... argumentTypes);
```

Получает указанный метод, публичный или непубличный, который имеет аргументы, совпадающие с типами, перечисленными в `argumentTypes`, и который объявлен в этом классе (унаследованные методы не считаются).

```
Constructor [] getConstructors();
```

Получает все публичные конструкторы этого класса.

```
Constructor getConstructor(Class ... argumentTypes);
```

Получает указанный публичный конструктор этого класса, который имеет аргументы, совпадающие с типами, перечисленными в `argumentTypes`.

```
Constructor [] getDeclaredConstructors();
```

Получает все публичные и непубличные конструкторы этого класса.

```
Constructor getDeclaredConstructor(Class ... argumentTypes);
```

Получает указанный конструктор, публичный или непубличный, который имеет аргументы, совпадающие с типами, перечисленными в `argumentTypes`.

```
Class [] getDeclaredClasses();
```

Получает публичные и непубличные внутренние классы, объявленные в этом классе.

```
Constructor [] getInterfaces();
```

Получает все интерфейсы, реализованные этим классом в том порядке, в котором они объявлены.

Как видите, методы `getMethod()` и `getConstructor()` пользуются преимуществом списка аргументов переменной длины в Java, чтобы позволить вам передавать типы аргументов. В более старых версиях Java вам приходилось передавать массив типов `Class` на их место. Мы покажем пример позже.

В качестве быстрого примера мы покажем, как быстро перечислять все публичные методы класса `java.util.Calendar`:

```
for ( Method method : Calendar.class.getMethods() )
System.out.println( method );
```

Здесь мы использовали запись `.class`, чтобы получить ссылку к объекту класса `Class` класса `Calendar`. Вспомните разговор о классе `Class`; методы отражения не принадлежат конкретному экземпляру самого класса `Calendar`; они принадлежат объекту `java.lang.Class`, который описывает класс `Calendar`. Если бы мы хотели начать с экземпляра `Calendar` (или, скажем, неизвестного объекта), мы могли бы использовать метод объекта `getClass()` вместо этого:

```
Method [] methods = myUnknownObject.getClass().getMethods();
```

Модификаторы и безопасность

Все типы членов класса Java — поля, методы, конструкторы и внутренние классы — имеют метод `getModifiers()`, который возвращает набор меток, указывающий является ли член доступным на приватном, защищенном, дефолтном уровне или он доступен публично. Вы можете проверить это при помощи `java.lang.reflect.Modifier` таким образом:

```
Method method = Object.class.getDeclaredMethod( "clone" ); // нет
аргументов
int perms = method.getModifiers();
System.out.println( Modifier.isPublic( perms ) ); // ложь
System.out.println( Modifier.isProtected( perms ) ); // правда
System.out.println( Modifier.isPrivate( perms ) ); // ложь
```

В этом примере метод `clone()` в классе `Object` является защищенным.

Доступ к API отражения регулируется диспетчером безопасности. Полностью доверенное приложение имеет доступ ко всей описанной ранее функциональности; он может получить доступ к членам классов на уровне ограничения нормально предоставленного кода в его рамках. Однако также возможно получить специальный доступ к коду, так что он сможет использовать API отражения, чтобы получить доступ к приватным и защищенным методам других классов таким образом, который язык Java обычно не позволяет.

Классы `Field`, `Method`, и `Constructor` расширяются от класса, называемого `AccessibleObject`. Класс `AccessibleObject` имеет один важный метод, называемый `setAccessible()`, что позволяет вам выключать обычную безопасность при доступе к отдельным членам класса. Это может показаться чересчур простым. И это действительно просто, но позволит ли вам этот метод отключить безопасность — это функция диспетчера безопасности Java и политики безопасности. Вы можете делать это в обычном приложении Java, работая без политики безопасности, но, например, не в апплете или другой безопасной среде. Например, чтобы использовать метод `clone()` класса `Object`, все, что нам нужно сделать (при условии, что нет противоречия с диспетчером безопасности), это:

```
Method method = Object.class.getDeclaredMethod( "клонировать" );
method.setAccessible( true );
```

Доступ к полям

Класс `java.lang.reflect.Field` представляет статические переменные и переменные экземпляра. `Field` имеет полный набор перегруженных методов доступа для всех базовых типов (например, методы `getInt()` и `setInt()`, `getBoolean()` и `setBoolean()`) и `get()` и `set()` для доступа к полям, которые являются типами ссылок. Давайте рассмотрим этот класс:

```
class BankAccount {
public int balance;
}
```

При помощи API отражения мы можем читать и изменять значения баланса публичного целочисленного поля:

```
BankAccount myBankAccount = ...;
...
```

```
try {
Field balanceField = BankAccount.class.getField("balance");
// прочесть это
int mybalance = balanceField.getInt( myBankAccount );
// изменить это
balanceField.setInt( myBankAccount, 42 );
} catch ( NoSuchFieldException e ) {
... // в этом классе нет поля "balance"
} catch ( IllegalAccessException e2) {
... // у нас нет разрешения для доступа к этому полю
}
```

В этом примере предполагается, что мы уже знаем структуру объекта `BankAccount`. В общем, мы можем собрать эту информацию от самого объекта.

Все методы доступа к данным `Field` ссылаются на определенный экземпляр объекта, к которому мы хотим обратиться. В этом примере метод `getField()` возвращает объект `Field`, который представляет поле `balance` класса `BankAccount`.

Этот объект не ссылается ни к какому специфическому `BankAccount`. Следовательно, чтобы прочитать или изменить любой специфический `BankAccount`, мы вызываем `getInt()` и `setInt()` при помощи ссылки на `myBankAccount` — он является отдельным экземпляром объекта, содержащим поле, с которым мы хотим работать. Для статического поля мы бы использовали здесь значение `null`. Если мы попытаемся обратиться к полю, которое не существует, или если у нас нет должного разрешения на чтение или запись в этом поле, возникает исключение. Если мы сделаем поле `balance` публичным, мы все еще можем искать объект `Field`, который описывает его, но мы не сможем прочесть или записать его значение.

Таким образом, мы не делаем ничего такого, что мы не могли бы сделать со статическим кодом во время компиляции; пока `balance` является публичным членом класса, к которому мы хотим обратиться, мы можем писать код, чтобы читать или менять его значение. Важно то, что мы обращаемся к `balance` во время выполнения, и мы могли бы точно также использовать эту технологию, чтобы исследовать поле `balance` в классе, который динамически загружался или который мы только что обнаружили, осуществляя итерации по полям объекта при помощи метода `getDeclaredFields()`.

Доступ к методам

Класс `java.lang.reflect.Method` представляет статический метод, или метод экземпляра. Подчиняясь обычным правилам безопасности, метод `invoke()` объекта `Method` может использоваться для вызова методов лежащего в основе объекта с указанными аргументами. Да, в Java действительно есть что-то похожее на указатель метода!

Как пример мы напишем приложение Java под названием `Invoke`, которое берет имя класса Java и имя вызываемого метода в качестве аргументов в командной строке. Для упрощения предположим, что метод статический и не имеет аргументов (безусловное ограничение):

```
//файл: Invoke.java
import java.lang.reflect.*;
class Invoke {
public static void main( String [] args ) {
try {
Class clas = Class.forName( args[0] );
Method method = clas.getMethod( args[1] );
// названный метод,
// без аргументов
Object ret = method.invoke( null );
// вызов статического метода
System.out.println(
"Invoked static method: " + args[1]
+ " of class: " + args[0]
+ " with no args\nResults: " + ret );
} catch ( ClassNotFoundException e ) {
// Class.forName() не может найти класс
} catch ( NoSuchMethodException e2 ) {
// этот метод не существует
} catch ( IllegalAccessException e3 ) {
// у нас нет разрешения для вызова этого метода
} catch ( InvocationTargetException e4 ) {
// возникло исключение при вызове этого метода
System.out.println(
"Method threw an: " + e4.getTargetException() );
}
}
}
```

Мы можем запустить invoke, чтобы вызвать значение часов системы:

```
% java Invoke java.lang.System currentTimeMillis
Invoked static method: currentTimeMillis of class:
java.lang.System with no args
Results: 861129235818
```

Нашим первым заданием будет поиск указанного класса Class по имени. Чтобы это сделать, мы вызываем метод forName() с именем желаемого класса (первый аргумент командной строки). Затем мы запрашиваем указанный метод по его имени. getMethod() имеет два аргумента: первым является имя метода (второй аргумент командной строки), а вторым массив объектов Class, который уточняет сигнатуру метода. (Помните, что любой метод может быть перегружен; вы должны указать сигнатуру, чтобы дать понять, какая версия вам нужна.) Поскольку наша простая программа вызывает только методы без аргументов, мы создаем анонимный пустой массив объектов класса Class. Если бы мы хотели вызвать метод, который принимает аргументы, мы бы передали массив классов их соответствующих типов в нужном порядке. Для простых типов мы бы использовали стандартные обертки (Integer, Float, Boolean, и др.) для хранения значений. Классы простых типов в Java представлены специальными статическими полями TYPE их соответствующих оберток; например, используйте Integer.TYPE для класса Class для типа int. Как показано в комментариях в коде, начиная с Java 5.0, методы getMethod() и invoke() принимают списки аргументов переменной длины, и это означает, что мы вообще можем пропустить аргументы и Java создаст нам пустой массив.

Когда у нас есть объект Method, мы вызываем метод invoke(). Это вызывает наш целевой метод и возвращает результат как объект класса Object. Чтобы сделать что-то нетривиальное с этим объектом, вы должны привести его к чему-то более точному. Предположительно, поскольку мы вызываем метод, мы знаем, какого объекта ожидать. Но если бы мы не знали, мы могли бы использовать метод MethodgetReturnType(), чтобы получить класс Class возвращаемого типа во время выполнения. Если возвращаемое значение является простым типом, таким как boolean, оно будет обернуто в стандартный класс-обертку для этого типа. Если этот метод возвращает void, invoke() возвращает объект java.lang.Void. Это класс-«обертка», который представляет возвращаемые значения void.

Первым объектом в invoke() является объект, для которого мы хотели бы вызвать метод. Если метод статический, объекта нет, поэтому мы ставим первому аргументу значение null. Это приведение типа в на-

шем примере. Второй аргумент является массивом объектов, которые должны передаваться как аргументы методу. Их типы должны совпадать с типами, указанными в вызове метода `getMethod()`. Поскольку мы вызываем метод без аргументов, мы можем передать `null` второму аргументу в `invoke()`. Как и в случае со значением возвращаемого типа, вы должны использовать классы-обертки для простых типов аргументов.

Исключения, показанные в предыдущем коде, возникают, если мы не можем найти разрешение на доступ к методу или у нас его нет. Дополнительно исключение `InvocationTargetException` возникает, если вызываемый метод сам выбрасывает исключение какого-то типа. Вы можете найти, что он выбросил, вызвав метод `getTargetException()` исключения `InvocationTargetException`.

Доступ к конструкторам

Класс `java.lang.reflect.Constructor` представляет конструктор объектов, так же как класс `Method` представляет метод. Вы можете использовать его, подчиняясь диспетчеру безопасности, конечно, чтобы создать новый экземпляр объекта, даже с конструкторами, которые требуют аргументы. Вспомните, что вы можете создавать экземпляры класса при помощи `Class.newInstance()`, но не можете указать аргумент с этим методом. Если вам действительно нужно это сделать, то вот и решение проблемы. Здесь мы создадим экземпляр `java.util.Date`*, передав строчный аргумент конструктору:

```
try {
Constructor<Date> cons =
Date.class.getConstructor( String.class );
Date date = cons.newInstance( "Jan 1, 2006" );
System.out.println( date );
} catch ( NoSuchMethodException e ) {
// getConstructor()не смог найти конструктор, который мы описали
} catch ( InstantiationException e2 ) {
// класс является обстрактным
} catch ( IllegalAccessException e3 ) {
// у нас нет разрешения на создание экземпляра
} catch ( InvocationTargetException e4 ) {
// конструкция выбросила исключение
}
```

* Этот конструктор `Date` является устаревшим, но послужит нам примером.

История очень похожа на вызов метода; в конце концов, конструктор в действительности — это не что иное, как метод с некими странными свойствами. Мы ищем подходящий конструктор для нашего класса `Date`, — тот, который принимает один объект класса `String` в качестве аргумента, передавая тип `String.class` методу `getConstructor()`. Здесь мы используем синтаксис аргумента переменной из Java 5.0. Если бы конструктор требовал больше аргументов, мы бы передали дополнительные объекты `Class`, представляющие классы каждого аргумента. Затем мы можем вызвать метод `newInstance()`, передавая ему соответствующий объект аргумента. Опять же, чтобы передать простые типы, мы сначала обернули бы их в их типы оберток. В конце концов мы выведем результирующий объект в `Date`. Отметьте, что мы незаметно пробрались в другую странную конструкцию, используя обобщения. Тип конструктора `<Date>` здесь просто позволяет нам указать класс `Constructor` для типа `Date`, ослабив потребность в приведении типов для результата метода `newInstance()`, как ранее.

Исключения из предыдущего примера применяются здесь также вместе с исключениями `IllegalArgumentException` и `InstantiationException`. Последнее выбрасывается, если класс объявлен как `abstract` и, следовательно, не может быть инициализирован.

Что насчет массивов?

API отражения позволяет вам создавать и анализировать массивы базовых типов, используя класс `java.lang.reflect.Array`. Процесс точно такой же, как и с другими классами, поэтому мы не будем рассматривать его в подробностях. Первоочередное свойство — это статический метод класса `Array`, называемый методом `newInstance()`, который создает массив, позволяющий вам указать базовый тип и длину. Вы также можете использовать его, чтобы создать многомерный массив экземпляров, указав массив длин (одна для каждой величины). Для более подробной информации к вашей любимой справке по языку Java.

Доступ к информации обобщенного типа

В главе 8 мы рассмотрим обобщения, которые впервые появились в Java 5.0. Обобщения — это важное добавление, которое привносит новые измерения (буквально) в концепцию типов в языке Java. С добавлением обобщений типы в классах и интерфейсах Java больше не являются

однозначными, но могут параметризироваться на один или более типов для создания нового, обобщенного типа. Чтобы еще больше все запутать, эти новые типы в действительности не генерируют новые классы, но вместо этого являются артефактами компилятора. Чтобы хранить общую информацию, Java добавляет информацию скомпилированным файлам класса.

API отражения может согласовывать все это в основном через добавление нового класса `java.lang.reflect.Type`, который способен представлять обобщенные типы. Рассмотрение этого всего в деталях немного выходит за рамки этой книги, и поскольку мы дойдем до обобщений только в главе 8, сейчас мы не будет останавливаться на этой теме подробно. Однако следующие отрывки кода могут помощь вам позже, если вы решите более вдумчиво изучить информацию о доступе к типам обобщения:

```
// публичный интерфейс List<E> расширяет Collection<E> { ... }
TypeVariable [] tv = List.class.getTypeParameters();
System.out.println( tv[0].getName() ); // "E"
```

Этот отрывок получает параметр типа класса java.util.List и выводит его имя:

```
class StringList extends ArrayList<String> { }
Type type = StringList.class.getGenericSuperclass();
System.out.println( type ); //
// "java.util.ArrayList<java.lang.String>"
ParameterizedType pt = (ParameterizedType)type;
System.out.println( pt.getActualTypeArguments()[0] ); //
// "класс java.lang.String"
```

Этот второй отрывок получает объект `Type` для класса, который расширяет обобщенный тип, а затем выводит фактический тип, на который он был параметризирован.

Доступ к данным аннотаций

Позже в этой главе мы обсудим аннотации — средство, которое позволяет метаданным добавляться к классам, методам и полям Java. Аннотации могут быть выборочно сохранены в компилированных классах Java, и к ним можно обращаться через API отражения. Это один из нескольких предполагаемых способов использовать аннотации, позволяющих коду во время выполнения видеть метаданные и предоставлять

специальные сервисы для аннотированного кода. Например, свойство (поле или устанавливающий метод) объекта Java может быть аннотировано, чтобы указывать, что он ожидает от контейнерного приложения установки его значения или экспорта его в каком-то значении.

Рассмотрение этого в подробностях выходит за рамки нашей книги; однако использовать данные аннотации в API отражения довольно просто. Классы Java, а также объекты классов `Method` и `Field` имеют следующие пары методов (и некоторые другие связанные методы):

```
public <A extends Annotation> A getAnnotation(Class<A> annotationClass)
public Annotation[] getDeclaredAnnotations()
```

Эти методы (первый является обобщенным методом, как описано в главе 8) возвращают типовые объекты `java.lang.annotation.Annotation`, которые представляют метаданные.

Адаптеры с динамическим интерфейсом

В идеале отражение Java позволило бы нам делать во время выполнения все, что мы можем делать во время компиляции (не заставляя нас генерировать и компилировать источник в байт-коде). Но не в этом все дело. Хотя мы можем динамически загружать и создавать экземпляры объектов во время выполнения, используя метод `Class.forName()`, нет общего способа для создания новых типов объектов — для которых не существует никаких классов файлов — на лету.

Однако класс `java.lang.reflect.Proxy` делает шаг навстречу решению этой проблемы, позволяя создание объектов адаптера, которое реализует произвольные интерфейсы Java во время выполнения. Класс `Proxy` — это рубрика, которая может генерировать класс адаптера, реализуя любой интерфейс (или интерфейсы), который вы хотите. Когда методы вызываются для класса адаптера, они делегируются одному методу в назначенном объекте `InvocationHandler`. Вы можете использовать это, чтобы создать динамические реализации любого типа интерфейса во время выполнения и обрабатывать вызовы метода, где вы хотите. Например, используя класс `Proxy`, вы можете вводить все вызовы метода в интерфейс, впоследствии делегируя их «реальной» реализации. Этот тип динамического поведения важен для инструментов, работающих с компонентами Java, которые должны регистрировать приемники событий. (Мы снова упомянем это в главе 22.) Это также полезно в отношении широкого спектра проблем.

В следующем отрывке мы берем имя интерфейса и создаем посредника, реализующего интерфейс. Это выводит сообщение, когда вызываются любые методы интерфейса:

```
import java.lang.reflect.*;
InvocationHandler handler =
new InvocationHandler() {
Object invoke( Object proxy, Method method, Object[] args ) {
System.out.println(
"Method: {[QUOTE-REPLACEMENT]}+ method.getName() +"()"
+" of interface: "+ interfaceName
+ " invoked on proxy!"
);
return null;
}
};
Class clas = Class.forName( MyInterface );
MyInterface interfaceProxy =
(MyInterface)Proxy.newProxyInstance(
clas.getClassLoader(), new Class[] { class }, handler );
// использование MyInterface
myInterface.anyMethod(); // Метод: anyMethod() ... вызван на прокси!
```

Результирующий объект приводится к желаемому типу интерфейса. Он вызовет наш обработчик, когда вызывается любой из его методов.

Во-первых, мы создадим реализацию `InvocationHandler`. Это объект с методом `invoke()`, который берет вызываемый класс `Method` в качестве своего аргумента и массив объектов, представляющих аргументы вызова метода. Затем мы вызываем класс интерфейса, который собираемся реализовать, используя метод `Class.forName()`. В конце концов, мы просим прокси создать для нас адаптер, указывая тип интерфейсов (вы можете указать больше одного), который хотим реализовать, и обработчик для использования.

Ожидается, что метод `invoke()` вернет объект правильного типа для вызова метода. Если он вернет неверный тип, будет выброшено специальное исключение выполнения.

Любые простые типы в аргументах или в возвращаемом значении должны быть обернуты в подходящий класс-обертку. (Система выполнения снимет обертку возвращаемого значения, если это необходимо.)

Для чего подходит отражение?

Хотя отражение в каком-то смысле является закулисным свойством языка Java, ему находится все больше и больше важных применений. В этой главе мы упоминали, что отражение используется для доступа к аннотациям выполнения. В главе 22 мы посмотрим, как с помощью отражения динамически обнаружить возможности и свойства объектов JavaBean. Это очень специализированные приложения — но что может отражение сделать для нас в повседневных ситуациях?

Мы можем использовать отражение, чтобы вести себя так, как если бы Java имел динамический вызов метода и другие полезные способности; в главе 22 мы увидим динамический класс адаптера, который использует отражение, чтобы делать вызовы вместо нас. Однако как общая практика программирования, динамический вызов метода является плохой идеей. Одно из первоначальных свойств Java (и то, что отличает его от некоторых похожих языков) — это безопасность типов. В этом отношении вы от многого отказываетесь, когда погружаетесь в отражения. И хотя API отражения выполняется очень хорошо, но в целом все же не так быстро, как скомпилированный вызов методов. Больше подходят для использования отражения ситуации, где вам нужно работать с объектами, о которых вы не можете заранее знать. Отражение ставит Java на более высокую ступеньку среди языков программирования, открывая возможности для новых типов приложений. Как мы упомянули, оно делает возможным одно использование аннотаций Java во время выполнения, позволяя нам изучать классы, методы и поля на предмет метаданных. Еще одна важная и активно расширяющаяся сфера употребления отражения — это интеграция Java с языками сценариев. При помощи отражения вы можете писать интерпретаторы языка в Java, которые могут обращаться к полным API Java, создавать объекты, вызывать методы, изменять переменные и делать все остальное, что программа Java может делать во время компиляции, во время выполнения программы. Возможно, вы зайдете так далеко, что сможете реализовывать заново язык Java в Java, допуская полностью динамические программы, которые могут делать что угодно. Звучит безумно? Что ж, кто-то уже делал это — один из авторов этой книги! Мы объясним далее.

Язык сценариев BeanShell

Я (Пэт) не могу устоять перед тем, чтобы вставить свои пять копеек про BeanShell — мой бесплатный легковесный язык сценариев с откры-

тым кодом. BeanShell — это как раз то, на что мы ссылались в предыдущем разделе, — приложение Java, которое использует Reflection API, чтобы выполнять операторы и выражения Java динамически. Вы можете использовать BeanShell интерактивно, чтобы быстро попробовать некоторые примеры в этой книге, или пойти дальше и начать использовать синтаксис Java в качестве языка сценариев в ваших приложениях. BeanShell использует API отражения Java на полную и служит демонстрацией того, насколько среда выполнения Java в действительности является динамической.

Вы можете найти BeanShell на его сайте. Смотрите приложение B, чтобы получить больше информации о начале работы с BeanShell. Мы надеемся, что вы найдете его интересным и полезным!

Аннотации

Как мы упоминали в главе 4, Java долгое время поддерживала ограниченный тип метаданных исходного кода Java через использование тегов комментариев Javadoc. С такими тегами Javadoc, как `@deprecated` или `@author`, мы можем добавлять некоторую информацию классу, методу или полю, вставляя ее в комментарии над элементом. В этом случае информация в основном полезна генератору документации Javadoc, поскольку комментарии существуют только в исходном коде Java. Однако разработчики долгое время хотели получить способ употребления метаданных для других целей. И наконец действительно были разработаны некоторые инструменты, которые читают расширенные теги в стиле Javadoc в комментариях и делают с ними все, что угодно, включая генерацию кода и документацию. В Java 5.0 была добавлена формальная, расширяемая система метаданных, называемых аннотациями, которая предоставляет этот тип функционала на уровне источника, а также новые возможности для использования метаданных во время выполнения.

Аннотации позволяют вам добавлять метаданные в пакеты, классы, методы и поля Java. Эти метаданные могут быть утилизированы инструментами во время компиляции и по желанию сохранены в компилированных классах Java для использования также во время выполнения. Доступность аннотированных данных для работающих программ открывает новые возможности использования метаданных. Например, аннотации могут не только применяться во время

компиляции для генерации вспомогательных классов или ресурсов, но и использоваться сервером для обеспечения классам специальных сервисов, таких как импорт или экспорт значений, безопасность или мониторинг. Аннотации будут широко использоваться в Java XML Binding (JAXB), Java Servlets AP и веб-сервисах (JAX-WS), как мы увидим далее в книге. В этих случаях аннотации используются для упрощения конфигурации и информации о размещении. Формально, согласно спецификации, аннотации не предназначены для «прямого влияния на семантику программы». Однако это предупреждение является немного туманным и в сообществе Java существует некоторый страх, что эта способность откроет ящик Пандоры с потенциальным неправомерным применением. Надеемся, что разработчики будут использовать их сдержанно.

Только горстка «встроенных» аннотаций используется в Java повсеместно, и мы кратко сформулируем их в следующим разделе. Встроенные аннотации чаще используются в специализированных пакетах, таких как пакеты для веб-сервисов, а некоторые сами применяются для создания аннотаций. Создание собственных аннотаций синтаксически просто (по существу, как объявление интерфейса), но реализация поведения для них при помощи компилятора или системы выполнения немного выходит за рамки книги, поэтому мы не будем рассматривать эту тему здесь. JDK предоставляет инструмент фреймворка, называемый *apt*, который может использоваться для реализации аннотации на уровне источника, генерирующего и компилирующего код или ресурсные файлы во время компиляции. Обращение к данным аннотации во время выполнения производится при помощи API отражения, как было описано ранее в этой главе.

Использование аннотаций

Аннотации помещаются в код, предшествующий аннотируемому элементу при использовании символа `@`, за которым следует имя класса аннотации. Аннотация `@Deprecated` является примером самого простого типа, *аннотацией-маркером*, или *меткой-аннотацией*. Аннотация-маркер указывает на некую семантику просто своим присутствием. (В случае `@Deprecated` это означает, что член является устаревшим и компилятор должен генерировать предупреждение при его использовании.) Чтобы использовать аннотацию `@Deprecated`, мы помещаем ее перед классом, методом или полем `@Deprecated` таким образом:

```
@Deprecated
class OldClass { ... }
class AgingClass
{
@Deprecated
public someMethod() { ... }
...
}
```

В более общем смысле аннотации могут принимать «аргументы» в расширенном синтаксисе наподобие метода. Табл. 7.1 кратко формулирует возможные варианты.

Табл. 7.1. Использование аргументов в аннотации

Пример	Описание
`@Deprecated`	Аннотация-маркер (нет «данных»)
`@WarningMessage("Something about...")`	Один аргумент
`@TestValues( { "one", "two" } )`	Массив аргументов
`@Author( first="Pat", last="Niemeyer" )`	Аргументы с именем

Первая аннотация в таблице, `@Deprecated`, является настоящей аннотацией, как описано ранее; остальные три являются вымышленными. Чтобы принимать много значений, аннотация может использовать синтаксис массива с фигурными скобками (`{}`) или более новый синтаксис именных аргументов, показанный в последнем примере. Именной синтаксис позволяет передавать аргументы в любом порядке.

Аннотации на уровне пакета

Во введении мы упомянули, что пакеты Java не могут быть аннотированными. Это поднимает вопрос того, где нужно помещать такую аннотацию, поскольку обычно не существует места, где мы «объявляем» пакет Java; мы, как правило, просто используем их косвенно. Ответ заключается в том, что для удобства мы можем создать файл с именем *package-info.java* и поместить его в папку, соответствующую пакету Java. Файл не может содержать классы Java, но должен содержать оператор пакета. Аннотации пакета могут быть помещены в этот оператор пакета. В следующем примере мы исключаем пакет `learningjava.oldstuff`, так что использование любого из его классов генерирует предупреждение об исключении.

```
// файл: learningjava/service/package-info.java
/**
* Мы также можем поместить комментарии пакета здесь!
*/
@Deprecated
package learningjava.oldstuff;
```

Стандартные аннотации

Табл. 7.2 кратко описывает аннотации, поставляемые с Java.

Табл. 7.2. Стандартные аннотации

Аннотация	Описание
`@Deprecated`	Предупреждение об исключении члена
`@Override`	Указывает, что аннотируемые методы должны переопределять метод в родительском классе или будет выдано предупреждение компилятора
`@SuppressWarnings(value=" type ")`	Указывает, что типы указанного предупреждения должны быть запрещены компилятором для аннотируемого класса или метода

Мы уже обсудили аннотации `@Deprecated` и `@Override`, последний из которых мы рассмотрели в разделе «переопределяющие методы». Аннотация `@Suppress Warnings` предназначена для принужденного использования внедренного унаследованного кода с новым кодом при использовании обобщений в версиях после Java 5.0, но некоторые компиляторы могут не реализовывать это.

Дополнительные аннотации поставляются с Java как часть пакета `java.lang.annotations` и используются, чтобы аннотировать только другие аннотации (они на самом деле являются метааннотациями). Например, аннотация `java.lang.annotation.Retention` устанавливает политику хранения аннотации, указывая, была ли она сохранена в скомпилированном классе или загружена во время выполнения.

Инструмент apt

JDK Java поставляется с консольным инструментом обработки аннотации, `apt`, который является подобием фронтенда компилятора `javac`. `apt` и использует встраиваемые процессоры аннотаций, чтобы

обрабатывать аннотации в файлах источника перед тем, как код будет компилирован `javac`. Если вы пишете собственные аннотации на уровне источника, вы можете создать встраиваемый процессор аннотаций для `apt`, который будет вызываться для обработки ваших аннотаций в исходном коде. Ваш обработчик аннотаций может быть достаточно сложным, исследующим структуру исходного кода (в манере только для чтения) через предоставленное синтаксическое дерево (объектную модель) и генерирующим любые дополнительные файлы или действия. Если вы генерируете новые исходные файлы Java, они будут автоматически компилироваться для вас при помощи `javac`. Запуск `apt` для исходного файла без аннотаций просто проваливается в `javac`.

Глава 8
ОБОБЩЕНИЯ

Прошло 15 лет с появления языка программирования Java (и первой редакции этой книги). За это время язык возмужал и обрел заслуженный успех. Но до Java 5.0, шестого по счету важного выпуска Java, сам базовый язык значительным образом не менялся. Да, были незаметные перемены и ошибки в течение лет. Внутренние классы, добавленные очень рано, были важны. Но никакие улучшения в языке, которые происходили до этого момента, не влияли на ядро Java или всех разработчиков Java, как это сделали обобщенные типы в Java 5.0.

Обобщения ориентируются на абстракцию. Обобщения позволяют вам создавать классы и методы, которые работают одинаково для разных типов объектов. Понятие *общий* происходит от идеи, что мы хотели бы иметь возможность писать общие алгоритмы, которые могут широко использоваться заново во многих типах объектов, а не адаптировать наш код, чтобы он подходил под каждое условие. Эта идея не нова. Это сила, которая движет самим объектно-ориентированным программированием. Обобщения Java не столько добавляют новые возможности языку, сколько делают код Java, который можно использовать повторно, более легким для написания и чтения.

Обобщения переносят повторное использование на новый уровень, делая *тип* объектов, с которыми мы работаем, явным параметром общего кода. По этой причине на обобщения также ссылаются как на *параметризированные типы*. В случае общего класса разработчик указывает тип как параметр (аргумент), когда он использует общий тип. Класс параметризируется предоставленным типом, к которому адаптируется сам код.

В других языках к обобщениям иногда ссылаются как к *шаблонам*, что больше является понятием реализации. Шаблоны похожи на промежуточные классы, ожидающие свои параметры типа, и тогда они могут использоваться. Java идет по другому пути, имеющему и достоинства, и недостатки, которые мы опишем в подробностях в этом разделе.

Можно много говорить об обобщениях в Java. Некоторые трудные вопросы могут показаться сначала немного неясными, но не впадайте в уныние. Большая часть того, что вы будете делать с обобщениями, является легкой и интуитивной. Остальное придет, если набраться немного терпения и упорства.

Мы начнем наше обсуждение с самого важного случая в обобщениях: классы контейнеров и библиотеки. Далее мы сделаем шаг назад и узнаем все хорошее, плохое и ужасное о том, как работают обобщения в Java, перед тем как перейдем к деталям написания классов обобщений. Затем мы введем методы обобщений, которые грамотно получают их параметризированные типы, основанные на том, как они вызываются. Мы завершим тем, что рассмотрим пару настоящих обобщенных классов в API Java.

Контейнеры: новый уровень абстракций

В таких объектно-ориентированных языках программирования, как Java, полиморфизм означает, что объекты всегда в какой-то степени взаимозаменимы. Каждый дочерний элемент типа объекта может служить вместо его родительского типа и, в конце концов, каждый объект является дочерним объектом `java.lang.Object`, так сказать объектно-ориентированной «Евы». Следовательно, для большинства обобщенных типов контейнеров в Java естественно работать с типом `Object`, так что они могут хранить почти все, что угодно. Под контейнерами мы подразумеваем классы, которые содержат экземпляры других классов в какой-то степени. Фреймворк библиотек Java (Java Collections Framework) является лучшим примером контейнеров. Например, `List` содержит упорядоченную библиотеку элементов типа `Object`. `Map` содержит объединение пар «ключ/значение», где ключи и значения также являются самым общим типом — `Object`. С небольшой помощью оберток для простых типов эта классификация хорошо служит нам. Но (чтобы вы уж не слишком ощущали дзен) в каком-то смысле «библиотека любого типа» также является «библиотекой никакого типа», и работа с типами `Object` переносит большую часть ответственности на пользователя контейнера.

Это как костюмированная вечеринка для объектов, где все носят одинаковые маски и исчезают в толпе библиотеки. Когда объекты одеты как тип `Object`, компилятор не может больше видеть реальные типы и те-

ряет их след. Нарушить затем анонимность объектов, используя приведение типов, — обязанность пользователя. И как при попытке сдернуть фальшивую бороду завсегдатая вечеринки, вам лучше иметь корректное приведение типов, или вы получите неприятный сюрприз.

```
Date date = new Date();
List list = new ArrayList();
list.add( date );
...
Date firstElement = (Date)list.get(0); // Верно ли приведение типов?
Возможно.
```

Интерфейс `List` имеет метод `add()`, который принимает любой тип класса `Object`. Здесь мы назначили экземпляр `ArrayList`, который является просто реализацией интерфейса `List`, и добавили объект `Date`. Является ли приведение типов в этом примере корректным? Зависит от того, что произойдет через опущенный период времени «...».

Можно ли исправить контейнеры?

Естественно будет спросить, существует ли способ улучшить ситуацию. Что если мы знаем, что только собираемся поместить объекты `Date` в наш список? Можем ли мы создать собственный список, который принимает только объекты `Date`, избавиться от приведения типов и позволить компилятору снова нам помочь? Возможно, это удивительно, но ответ «нет». По крайней мере, не удовлетворяющим образом.

Нашим первым побуждением может быть попытка «переопределить» методы `ArrayList` в подклассе. Но, конечно, перезаписывание метода `add()` в подклассе в действительности ничего бы не переопределило; оно бы добавило новый перегруженный метод.

```
public void add( Object o ) { ... }
public void add( Date d ) { ... } // перегруженный метод
```

Результирующий объект до сих пор принимает любой тип объекта, он просто вызывает другие методы для этого.

Продвигаясь дальше, мы, возможно, примем вызов большего задания. Например, мы можем написать наш собственный класс `DateList`, который не расширяет `ArrayList`, а делегирует содержимое его метода реализации класса `ArrayList`. После огромной трудоемкой работы это

даст нам объект, который делает то же, что делает `List`, но работает с классами `Date`. Однако сейчас мы навредили самим себе, поскольку наш контейнер больше не является реализацией `List` и мы не можем использовать его взаимозаменяемо со всеми утилитами, которые имеют дело с библиотеками, такими как `Collections.sort()`, или добавить к другой библиотеке при помощи метода `addAll()` класса `Collection`. Если обобщить, проблема состоит в том, что вместо улучшения поведения наших объектов в действительности хотим поменять их контракт с пользователем. Мы хотим адаптировать их API до более определенного типа, а полиморфизм этого не разрешает. Может показаться, что мы застряли с классами `Object` для наших библиотек. Именно здесь в игру вступают обобщения.

Ввод обобщений

Обобщения являются улучшением синтаксиса классов, которое позволяет нам специализировать класс для данного типа или набора типов. Общий класс требует одного или более *параметров типа*, когда мы обращаемся к типу класса, и использует их, чтобы настроить себя.

Если вы посмотрите на источник или Javadoc для класса `List`, например, вы увидите, что он определен каким-то таким образом:

```
public class List< E > {
...
public void add( E element ) { ... }
public E get( int i ) { ... }
}
```

Идентификатор E между угловыми скобками (<>) является *типизированной переменной*. Он указывает, что класс `List` является обобщенным и требует тип Java в качестве аргумента, чтобы сделать его полным. Имя E является произвольным, но существуют соглашения, которые мы увидим по мере продвижения. В этом случае типизированная переменная E представляет тип элементов, который мы хотим хранить в списке. Класс `List` ссылается к типизированной переменной внутри его тела и методов, как если бы это был реальный тип, который будет подменен позже. Переменная типа может использоваться для объявления переменных экземпляра, аргументов методов, и возвращаемого типа методов. В этом случае E используется как тип элементов которые мы будем

добавлять при помощи метода `add()` и возвращаемого типа метода `get()`. Давайте посмотрим, как его использовать.

Тот же синтаксис угловых скобок поставляет параметр типа, когда мы хотим использовать тип `List`:

```
List<String> listOfStrings;
```

В этом отрывке мы объявили переменную, которая называется `listOfStrings`, используя общий тип `List` с параметром типа `String`. `String` ссылается к классу `String`, но мы могли бы специализировать `List` любым другим типом класса. Например:

```
List<Date> dates;
List<java.math.BigDecimal> decimals;
List<Foo> foos;
```

Завершение типа, обеспечивая его тип параметра, называется *реализацией типа*. Иногда это также называется *вызовом типа* по аналогии с вызовом метода и обеспечением его аргументами. Тогда как с обычным типом Java мы просто обращаемся к типу по имени, общий тип должен быть реализован с параметрами всегда, когда он используется*. Так, это означает, что мы должны реализовывать тип везде, где могут возникнуть типы, как объявленный тип переменной (что показано в этом отрывке кода), как тип аргумента метода или в выражении размещения объекта при использовании ключевого слова `new`.

Возвращаясь к нашему `listOfStrings`, то, что мы сейчас имеем, это фактически `List`, в котором тип `String` заменен типизированной переменной `E` в теле класса:

```
public class List< String > {
...
public void add( String element ) { ... }
public String get( int i ) { ... }
}
```

У нас есть специальный класс `List` для работы с элементами типа `String` и *только* с элементами типа `String`. Этот тип метода не может принимать произвольные типы объектов.

---

* То есть если вы не хотите использовать общий тип не общим способом. Мы поговорим о «необработанных» типах позже в этой главе.

List — это просто интерфейс. Чтобы использовать переменную, нам нужно создать экземпляр объекта одной из реализаций класса List. Мы будем использовать ArrayList, как уже делали во вступлении. Как и раньше, ArrayList — это класс, который реализует интерфейс List, но в этом случае оба класса List и ArrayList являются общими классами. По существу, они требуют типовые параметры для указания им, где они используются. Конечно, мы создадим наш ArrayList для хранения элементов типа String, чтобы соответствовать нашему классу строк List:

```
List<String> listOfStrings = new ArrayList<String>
List<String> listOfStrings = new ArrayList<>(); // Или сокращение в Java
7.0
// и позже
```

Как обычно, ключевое слово new принимает тип Java и величины с возможными аргументами конструктора класса. В этом случае типом является ArrayList<String> — общий тип ArrayList, реализованный с типом String. Объявление переменных, как показано в первой строке предыдущего примера, немного громоздкое, поскольку требует типизировать общий тип параметра дважды (один раз слева в типе переменной и один раз справа в инициализирующем выражении). И в сложных случаях обобщенные типы могут становиться очень длинными и вложенными один в другой. В Java 7 компилятор достаточно умен, чтобы угадать тип инициализирующего выражения по типу переменной, к которой вы его назначаете. Это называется *интерфейс обобщенного типа* и сводится к факту, что вы можете сократить правую часть объявления вашей переменной, оставив содержимое обозначения <>, как показано во второй строке примера.

Теперь мы можем использовать наш специализированный тип List со строками. Компилятор защищает нас от попытки просто поместить что-то отличное от объекта String (или подтипа String, если там был такой) в список и позволяет нам вызывать их при помощи метода get(), не требуя приведения типов:

```
List<String> listOfStrings = new ArrayList<String>();
listOfStrings.add("eureka! ");
String s = listOfStrings.get(0); // "эврика! "
listOfStrings.add( new Date() ); // Ошибка во время компиляции!
```

Давайте возьмем другой пример из API библиотек. Интерфейс Map предоставляет преобразование наподобие словаря, которое ассоциирует

ключевые объекты с объектами значений. Ключи и значения не должны быть одного и того же типа. Общий интерфейс `Map` требует два параметра типа: один для типа ключа и один для типа значения. Javadoc выглядит таким образом:

```
public class Map< K, V > {
...
public V put( K key, V value ) { ... } // возвращает любое старое значение
public V get( K key ) { ... }
}
```

Мы можем создать интерфейс `Map`, который будет хранить объекты класса `Employee` по «идентификационному номеру работника» — номеру типа `Integer` — таким образом:

```
Map< Integer, Employee > employees = new HashMap< Integer, Employee >();
Integer bobsId = ...;
Employee bob = ...;
employees.put( bobsId, bob );
Employee employee = employees.get( bobsId );
```

Здесь мы использовали класс `HashMap`, который является общим классом, реализующим интерфейс `Map`, и реализовали оба типа с параметрами типа `Integer` и `Employee`. Теперь `Map` работает только с ключами типа `Integer` и хранит значения типа `Employee`.

Причина, по которой мы использовали здесь `Integer` для хранения наших типов, заключается в том, что параметры типа общего класса должны быть типами класса. Мы не можем параметризировать общий класс примитивным типом, таким как `int` или `boolean`. Но благодаря автоматическому распаковыванию простых типов в Java (смотрите главу 5), мы как будто почти можем это делать, т. к. оно позволяет нам использовать простые типы, как если бы они были типами оберток:

```
employees.put( 42, bob );
Employee bob = employees.get( 42 );
```

Здесь автоупаковка дважды конвертировала целое число 42 в обертку типа `Integer`.

В главе 11 мы увидим, что все классы и интерфейсы библиотеки Java являются общими. Более того, дюжины других API используют обобщения, чтобы позволит вам адаптировать их под определенные типы. Мы поговорим о них по мере их появления в книге.

Разговор о типах

Перед тем как мы перейдем к более важным вещам, нужно сказать несколько слов о том, как мы описываем отдельную параметризацию общего класса. Поскольку наиболее распространенный случай для обобщений — это объекты, похожие на контейнер, принято думать, что общий тип «хранит» тип параметра. В нашем примере мы назвали наш `List<String>` списком строк, потому что, без сомнения, это то, чем он является. Таким же образом мы могли назвать нашу карту сотрудников «Карта идентификационных номеров сотрудников относительно объектов класса `Employee`». Однако эти описания фокусируются немного больше на том, что делают классы, чем на самом типе. Возьмите вместо этого один контейнер объекта, называемый `Trap< E >`, который может реализовываться на объекте типа `Mouse` или типа `Bear`; то есть `Trap<Mouse>` или `Trap<Bear>`. Нашим первым порывом будет назвать новый тип «мышеловка» (mouse trap) или «ловушка на медведей» (bear trap). Подобным образом мы могли бы подумать о нашем списке строк как о новом типе — «список строк» или нашей карте сотрудников как о новом типе «целочисленной `объектной` карты сотрудников». Вы можете использовать любое текстовое наполнение, которое хотите, но эти последние описания больше фокусируются на понятии обобщения как *типа* и могут помочь немного позже, когда мы будем рассматривать то, как обобщенные типы связаны в системе типов. Там мы увидим, что терминология контейнера оказывается немного нелогичной.

В следующем разделе мы продолжим рассматривать обобщенные типы в Java с разных углов зрения. Мы уже немного видели, что они могут делать; теперь нам нужно поговорить о том, как они это делают.

«Ложки не существует»

В фильме «Матрица»* герою Нео предлагают выбор. Взять синюю таблетку и остаться в мире фантазий или взять красную таблетку и увидеть вещи такими, какие они есть на самом деле. Когда мы имеем дело с обобщениями в Java, мы сталкиваемся с похожей онтологической дилеммой. В любом разговоре об обобщениях мы можем дойти только до того момента, пока не столкнемся с тем, как они в действительности реа-

* Для тех из вас, кто хочет узнать контекст заголовка этой главы, вот откуда он: Мальчик: «Не пытайся согнуть ложку. Это невозможно. Вместо этого просто попытайся понять правду». Нео: «Какую правду?» Мальчик: «Никакой ложки нет». Нео: «Никакой ложки нет?» Мальчик: «Тогда ты увидишь, что гнется не ложка, а ты». (фильм Энди и Ларри Вачовски «Матрица», Warner Brothers, 1999).

лизуются. Наш мир фантазий — это мир, созданный компилятором, чтобы сделать создаваемый нами код более легким для восприятия. Наша реальность (хоть и не такой мрачный кошмар, как в кино) является более суровым местом, наполненным невиданными опасностями и вопросами. Почему приведения типов и тесты не работают надлежащим образом с обобщениями? Почему я не могу применить то, что является двумя разными общими интерфейсами в одном классе? Почему я могу объявить массив обобщенных типов, хоть в Java нет способа создания таких массивов? Мы ответим на все эти вопросы и даже больше в этом разделе, и вам не придется ждать продолжения. Давайте начнем.

Цели разработки обобщений в Java были значительными: добавить радикально новый синтаксис в язык, который безопасно вводит параметризированные типы, не влияя на выполнение и, ах, кстати, делает его обратно совместимым со всем существующим кодом Java и серьезно не меняет компилированные классы. Действительно, просто поразительно, что все эти условия вообще могут быть удовлетворены, и не удивительно, что это заняло так много времени. Но, как всегда, были нужны компромиссы, которые ведут к некоторым проблемам.

Чтобы выполнить этот трюк, Java пользуется услугами технологии, называемой *стирание*, которая связана с идеей, что поскольку почти все, что мы делаем с обобщениями, применяется статически во время компиляции, общую информацию не нужно переносить в компилированные классы. Общая природа классов, приводимая в жизнь компилятором, может быть «стерта» в компилированных классах, что позволяет нам поддерживать совместимость с кодом, не являющимся общим. Поскольку Java сохраняет информацию об общих свойствах классов в компилированной форме, эта информация в основном используется компилятором. Время выполнения Java вообще ничего не знает об обобщениях.

Стирание

Давайте посмотрим на скомпилированный общий класс: наш друг `List`. Мы легко можем сделать при помощи команды `javap`:

```
% javap java.util.List
public interface java.util.List extends java.util.Collection{
...
public abstract boolean add(java.lang.Object);
public abstract java.lang.Object get(int);
```

Результат выглядит так же, как выглядел до обобщений Java, как вы можете подтвердить любой старшей версией JDK. Причем тип элементов, используемый с методами `add()` и `get()`, является типом `Object`. Теперь вы, возможно, подумаете, что это всего лишь уловка и что когда реализуется фактический тип, Java создает новую версию класса внутренне. Но не в этом дело. Это единственный класс `List`, и это фактический тип выполнения, используемый параметризациями `List`; например, `List<Date>` и `List<String>`, как мы можем подтвердить:

```
List<Date> dateList = new ArrayList<Date>();
System.out.println( dateList instanceof List ); // правда!
```

Но наш общий класс `dateList` не реализует методы класса `List`, которые мы сейчас обсуждали:

```
dateList.add( new Object() ); // Ошибка во время компиляции!
```

Это иллюстрирует немного противоречивую природу обобщений в Java. Компилятор верит в них, но система выполнения говорит, что они являются иллюзией. Что если мы попробуем что-то немного более разумное и просто проверим, что `dateList` является `List<Date>`:

```
System.out.println( dateList instanceof List<Date> ); // Ошибка во время
компиляции!
// нелегальный, общий тип для instanceof
```

В этот раз компилятор просто переворачивает все с ног на голову и говорит «Нет». Вы не можете проверить общий тип операцией `instanceof`. Поскольку нет фактических дифференцируемых классов для разных параметризаций `List` во время выполнения, нет способа, благодаря которому `instanceof` может определить отличие одной ипостаси `List` от другой. Вся проверка безопасности обобщения проведена во время компиляции, и теперь мы просто имеем дело только с фактическим типом `List`.

В действительности произошло то, что компилятор стер весь синтаксис угловых скобок и заменил переменные типа в нашем классе `List` типом, который может работать во время выполнения с любым разрешенным типом: в этом случае с типом `Object`. Может показаться, будто мы вернулись к тому, с чего начали, не считая того, что компилятор до сих пор владеет информацией, чтобы привести в исполнение наше использование обобщений в коде во время компиляции и, следовательно, справится с приведением типов для нас. Если вы декомпилируете класс, использующий `List<Date>` (команда `javap` с опцией `-c` покажет

вам байт-код, если вы захотите), вы увидите, что компилированный код в действительности содержит приведение типа к типу `Date`, хоть мы сами его и не писали.

Теперь мы можем ответить на один из вопросов, которые мы подняли в начале главы («Почему я не могу реализовать то, что является двумя разными общими интерфейсами в одном классе?»). Мы не можем иметь класс, который реализует две разных общих реализации типа `List`, потому что они в действительности являются одним и тем же типом во время выполнения и нет способа разделить их:

```
public abstract class DualList implements List<String>, List<Date> { }
// Ошибка: java.util.List не может наследоваться с разными аргументами
// <java.lang.String> и <java.util.Date>
```

Необработанные типы

Хотя компилятор относится к разным параметризациям обобщенного типа как к разным типам (с разными интерфейсами API) во время компиляции, мы видели, что только один настоящий тип существует во время выполнения. Например, класс `List<Date>` и `List<String>` делит простой старый класс `List`. `List` называется *необработанным типом* общего класса. Каждое обобщение имеет необработанный тип. Это вырожденная, «простая» форма Java, из которой удалена вся информация обобщенного типа, а переменные типа заменены таким общим типом Java как `Object`*.

До сих пор можно использовать необработанные типы в Java, как и до того, как в язык были добавлены обобщения. Единственным отличием является то, что компилятор Java генерирует предупреждение, когда они используются «небезопасным» образом. Например:

```
// Java-код без обобщений, использующий необработанный тип
List list = new ArrayList(); // назначение в порядке
list.add("foo"); // Предупреждение компилятора об использовании
необработанного типа
```

---

* Когда обобщения добавили в Java 5.0, все стало тщательно организовано таким образом, что необработанный тип общих классов был разработан, чтобы был точно такими же как ранее необобщенные типы. Поэтому необработанный тип List в Java 5.0 является таким же, как старый необобщенный тип List, который у нас был, начиная с JDK 1.2. Поскольку в основной части текущего кода Java в то время не использовались обобщения, эта эквивалентность и совместимость были очень важны.

Этот отрывок использует необработанный тип List, как использовал бы старомодный код до Java 5. Разница заключается в том, что теперь компилятор Java выдает *непроверенное предупреждение* о коде, если мы пытаемся вставить объект в список.

```
% javac MyClass.java
Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Компилятор дает нам инструкцию использовать опцию -Xlint:unchecked, чтобы получить более точную информацию о местоположении небезопасных операций:

```
% javac -Xlint:unchecked MyClass.java
warning: [unchecked] unchecked call to add(E) as a member of the raw type
java.util.
List: list.add("foo");
```

Заметьте, что создание и назначение необработанного `ArrayList` не генерирует предупреждение. Мы получаем предупреждение, только если пытаемся использовать «небезопасный» метод (который обращается к переменной типа). Это означает, что нормально использовать старомодные необобщенные API Java, которые работают с необработанными типами. Мы получаем предупреждения, когда делаем что-то небезопасное в нашем собственном коде.

И еще кое-что о стирании перед тем, как мы продолжим. В предыдущем примере переменные типа были заменены типом `Object`, который мог представлять любой тип, применимый к переменной типа `E`. Далее мы увидим, что дело не всегда в этом. Мы можем поместить ограничения, или *границы*, на типы параметра и, когда мы делаем это, компилятор может быть более строгим со стиранием типа. Мы объясним это подробнее позже, после того, как обсудим ограничения, но, например:

```
class Bounded< E extends Date > {
public void addElement( E element ) { ... }
}
```

В этом объявлении типа параметра говорится, что тип элемента E должен быть подтипом типа `Date`. В этом случае стирание метода `addElement()`, следовательно, является более ограничивающим чем `Object`, и компилятор использует тип `Date`:

```
public void addElement( Date element ) { ... }
```

Тип `Date` называется верхней границей этого типа, это означает, что он является здесь вершиной иерархии объектов и тип может реализовываться только на типе `Date` или на «более низких» (более производных) типах.

Теперь, когда мы имеем понятие о том, чем в действительности являются обобщенные типы, мы можем рассмотреть немного более подробно то, как они себя ведут.

Отношения параметризированных типов

Теперь мы знаем, что параметризированные типы делят общий, необработанный тип. Вот почему наш параметризированный тип `List<Date>` является просто типом `List` во время выполнения. В действительности мы можем назначить любую реализацию `List` необработанному типу, если мы захотим:

```
List list = new ArrayList<Date>();
```

Мы даже можем пойти другим путем и назначить необработанный тип отдельной реализации обобщенного типа:

```
List<Date> dates = new ArrayList(); // непроверенное предупреждение
```

Этот оператор генерирует непроверенное предупреждение на назначение, но впоследствии компилятор верит, что список содержал только типы `Date` до назначения. Также разрешено, хоть и бессмысленно, выполнять приведение типов в этом операторе. Мы поговорим о приведении типов в общих типах немного позже.

Какими бы ни были типы во время выполнения, компилятор командует парадом и не позволяет нам назначать то, что в действительности является несовместимым.

```
List<Date> dates = new ArrayList<String>(); // Ошибка во время компиляции!
```

Конечно, `ArrayList<String>` не реализует методы `List<Date>`, наколдованные компилятором, поэтому эти типы являются несочетающимися.

Но как насчет более интересных отношений типов? Интерфейс `List`, например, является подтипом более общего интерфейса `Collection`. Является ли отдельная реализация обобщенного типа `List` также назначаемой какой-то реализации общего интерфейса `Collection`? Зависит ли она от параметров типа и их отношений? Несомненно, `List<Date>` не яв-

ляется `Collection<String>`. Но является ли `List<Date>` `Collection<Date>`? Может ли `List<Date>` быть `Collection<Object>`?

Сначала разболтаем ответ, затем посмотрим пример. Правило заключается в том, что для простых типов общих реализаций, которые мы уже обсудили, *наследование применяется только к «базовому» общему типу, а не к типам параметров*. Более того, присваивание применяется только тогда, когда два общих типа реализуются *на одном и том же типе параметра*. Другими словами, все еще существует одномерное наследование, которое следует за базовым типом общего класса, но с дополнительным ограничением, что типы параметров должны быть идентичными.

Например, помня, что `List` является типом `Collection`, мы можем назначить реализации `List` реализациям `Collection`, когда параметр типа точно такой же:

```
Collection<Date> cd;
List<Date> ld = new ArrayList<Date>();
cd = ld; // Ok!
```

Этот отрывок кода говорит, что `List<Date>` является `Collection <Date>`, — довольно интуитивно. Но попытка применить такую же логику для варианта типов параметра терпит неудачу:

```
List<Object> lo;
List<Date> ld = new ArrayList<Date>();
lo = ld; // Ошибка компиляции! Несовместимые типы.
```

Хотя наша интуиция говорит нам, что типы `Date` в этом интерфейсе `List` могли бы жить счастливо, как типы `Object` в интерфейсе `List`, назначение является ошибкой. Мы объясним, почему так, в следующей главе, но сейчас просто отметьте, что типы параметров не являются точно такими же и нет отношения наследования между параметрами в обобщениях. Это тот случай, когда помогает рассмотрение реализации с точки зрения типов, а не того, что они делают. В действительности нет «списка дат» и «списка объектов», а скорее `DateList` и `ObjectList`, отношения между которыми не сразу очевидны.

Попытайтесь понять, что верно, а что нет в следующем примере:

```
Collection<Number> cn;
List<Integer> li = new ArrayList<Integer>();
cn = li; // Ошибка во время компиляции! Несовместимые типы.
```

Реализация интерфейса `List` может быть реализацией интерфейса `Collection`, но только если типы параметров абсолютно одинаковые. Наследование не следует за типами параметров, и этот пример не срабатывает.

Еще одно: ранее мы упоминали, что это правило применяется к простым типам реализаций, которые мы уже обсудили в этой главе. Какие же другие типы имеются в виду? Что ж, типы реализации, которые мы уже видели, когда вставляли в фактические типы Java в качестве параметра, называются *реализациями конкретного типа*. Позже мы поговорим об *обобщенных реализациях*, которые сходны с математическим набором операций над типами. Мы увидим, что возможно делать более экзотические реализации из обобщений, где отношения типов являются фактически двумерными, зависящими и от базового типа, и от параметризации. Но не беспокойтесь: это встречается не очень часто и на самом деле не так страшно, как звучит.

Почему `List<Date>` не является `List<Object>`?

Это разумный вопрос. Даже думая о типах `DateList` и `ObjectList`, мы все еще можем спрашивать, почему они не могут быть назначаемыми. Почему мы не можем назначить наш `List<Date>` как `List<Object>` и работать с элементами типа `Date` как с типами `Object`?

Причина возвращает нас к рациональному зерну обобщений, которое мы обсуждали в начале: смена API. В самом простом случае, если предположить, что тип `ObjectList` расширяет тип `DateList`, `DateList` имел бы все методы `ObjectList` и мы все еще могли бы вставлять типы `Object` в него. Теперь вы можете возразить, что обобщения позволяют нам изменять API, так что это больше неприменимо. Это правда, но есть проблема. Если бы мы могли назначить наш `DateList` переменной `ObjectList`, нам бы следовало суметь использовать методы `Object`, чтобы вставить в него элементы типов, отличные от типа `Date`. Мы могли бы *скрыть* `DateList` под `ObjectList` и попытаться обхитрить его принятием какого-то другого типа:

```
DateList dateList = new DateList();
ObjectList objectList = dateList; // В действительности невозможно сделать это
objectList.add( new Foo() ); // скорее всего, будет ошибка выполнения!
```

Мы ожидаем получить ошибку выполнения, если фактическая реализация `DateList` была представлена неверным типом объекта. Вот тут-

то нас и поджидает проблема. В обобщениях Java нет воспроизведения во время выполнения. Даже если эта функциональность была бы полезной, для текущей схемы Java нет иного способа узнать, что делать во время выполнения. Если взглянуть на это с другой стороны, это свойство просто опасно, поскольку позволяет ошибки во время выполнения, которые не могут быть отловлены во время компиляции. В общем, мы бы хотели отлавливать типовые ошибки во время компиляции. Запрещая эти назначения, Java может гарантировать, что ваш код является безопасным по отношению к типам, которые он компилирует без непроверенных предупреждений.

В действительности это последнее утверждение не является полностью верным, но оно не имеет отношения к обобщениям; оно имеет отношение к массивам. Если все это кажется вам знакомым, то потому, что мы упоминали это ранее по отношению к массивам Java. Типы массивов имеют иерархические отношения, которые позволяют возникнуть такому типу псевдонима:

```
Date [] dates = new Date[10];
Object [] objects = dates;
objects[0] = "not a date"; // Исключение ArrayStoreException во время
выполнения!
```

Однако массивы имеют представление во время реализации в качестве различных классов, и они проверяют себя во время выполнения, выбрасывая исключение `ArrayStoreException` только в этом случае. Поэтому в теории код Java не дает гарантий безопасности от компилятора по отношению к типам, если вы используете массивы таким образом.

Приведение типов

Сейчас мы поговорили об отношениях между общими типами и даже между общими и необработанными типами. Но мы еще не поднимали вопрос о понятии приведения типа. Необходимости в приведении типов не было, когда мы меняли обобщения с их необработанными типами. Вместо этого мы только что пересекли черту, которая запускает непроверенные предупреждения компилятора:

```
List list = new ArrayList<Date>();
List<Date> dl = list; // непроверенное предупреждение
```

Обычно мы используем приведение типов в Java для работы с двумя типами, которые могут быть назначаемыми. Например, мы можем попытаться привести `Object` к `Date`, поскольку возможно то, что тип `Object` является значением типа `Date`. Приведение типов тогда выполняет проверку во время выполнения, чтобы узнать, правы ли мы. Приведение между несвязанными типами является ошибкой во время компиляции. Например, мы даже не можем привести `Integer` к `String`. У этих типов нет отношений наследования. Что же насчет приведения типов между аналогичными общими типами?

```
Collection<Date> cd = new ArrayList<Date>();
List<Date> ld = (List<Date>)cd; // Ok!
```

Этот отрывок кода показывает правильное приведение типа более общего `Collection<Date>` к `List<Date>`. Приведение типов здесь является возможным, поскольку `Collection<Date>` назначаем от `List<Date>` и может фактически быть `List<Date>`. Подобным образом, следующее приведение типов отлавливает наши ошибки, где мы объявили `TreeSet<Date>` как `Collection<Date>` и попытались привести к `List<Date>`:

```
Collection<Date> cd = new TreeSet<Date>();
List<Date> ld = (List<Date>)cd; // Исключение ClassCastException во время
выполнения!
ld.add( new Date() );
```

Однако существует один случай, когда приведения типов не эффективны с обобщениями — когда мы пытаемся дифференцировать типы, основанные на их типах параметров:

```
Object o = new ArrayList<String>();
List<Date> ld = (List<Date>)o; // unchecked warning, ineffective
непроверенное предупреждение, неэффективно
Date d = ld.get(0); // опасно во время выполнения, косвенное приведение
типа может окончиться неудачей
```

Здесь мы использовали для `ArrayList<String>` простой псевдоним `Object`. Затем мы приводим его к `List<Date>`. К сожалению, Java не видит разницу между `List<String>` и `List<Date>` во время выполнения, поэтому приведение типов бесполезно.

Компилятор предупреждает нас об этом, генерируя непроверенное предупреждение в месте нахождения приведения типов; мы должны принять к сведению, что когда мы попытаемся использовать объект

приведения позже, то можем обнаружить, что он неверен. Приведения типов у обобщенных типов не эффективны во время выполнения, из-за стирания и недостатка информации о типах.

Написание обобщенных классов

Теперь, когда мы имеем (хотя бы какое-то) представление «конечного пользователя» об обобщениях, давайте попробуем сами написать несколько классов. В этой главе мы поговорим о том, как используются переменные типа в определении общих классов, где они могут возникнуть, и о некоторых их ограничениях. Мы также поговорим о наследовании обобщенных типов.

Переменная типа

Мы уже увидели основы того, как используются переменные типа в объявлении общего класса. Одна или больше переменных типа объявляются в угловых скобках (<>) и используются в теле и методы экземпляра в классе. Например:

```
class Mouse { }
class Bear { }
class Trap< T >
{
T trapped;
public void snare( T trapped ) { this.trapped = trapped; }
public T release() { return trapped; }
}
// использование
Trap<Mouse> mouseTrap = new Trap<Mouse>();
mouseTrap.snare( new Mouse() );
Mouse mouse = mouseTrap.release();
```

Здесь мы создали общий класс `Trap`, который может содержать любой тип объекта. Мы использовали переменную типа `T`, чтобы объявить переменную экземпляра типа параметра, также как в типе аргумента и возвращаемом типе двух методов.

Граница переменной типа — это часть с экземпляром класса, включающая методы и любые блоки инициализатора экземпляра. Статиче-

ская часть класса не нарушается общей параметризацией, и переменные типа не видны в статических методах или статических инициализаторах. Как вы можете догадаться, так же как все реализации обобщенного типа имеют только один действительный класс (необработанный тип), они также имеют только один общий контекст. Вы даже не можете вызвать статический метод через параметризированный тип. Вы должны использовать необработанный тип или экземпляр объекта.

Переменная типа может использоваться и в реализации типа других обобщенных типов, используемых классом. Например, если бы мы хотели, чтобы наш `Trap` содержал более одного животного, мы могли бы создать тип `List` для них в пределах класса, обращаясь к типу параметра таким образом:

```
List<T> trappedList = new ArrayList<T>();
```

Чтобы охватить всю базовую информацию, мы должны упомянуть, что реализации обобщенных типов для переменной типа ведут себя так же, как любой другой тип, и могут служить во всех местах, где могут другие реализации. Например, метод в нашем классе может принимать `List<T>` в качестве аргумента:

```
public void trapAll( List<T> list ) { ... }
```

Эффективный тип метода `trapAll()` в `Trap<Mouse>` тогда является просто:

```
trapAll( List<Mouse> list ) { ... }
```

Мы должны отметить, что это не то, что мы подразумеваем под *обобщенным методом*. Это просто обычный метод Java, который принимает общий тип в качестве аргумента. Мы поговорим о настоящих общих методах, которые могут *получать* их типы из аргументов и контексте назначения позже в этой главе. Переменная типа также может быть использована для параметризации общего родительского класса, как мы увидим в следующей главе.

Наследование обобщений

Из обобщенных типов можно создавать подтипы так же, как и из любого другого класса при помощи общих или не общих дочерних подклассов. Не родительский подкласс может расширить отдельную реа-

лизацию родительского типа, заполнив необходимые параметры, чтобы сделать ее конкретной:

```
class DateList extends ArrayList<Date> { }
DateList dateList = new DateList();
dateList.add( new Date() );
List<Date> ld = dateList;
```

Здесь мы создали не обобщенный подкласс `DateList` из конкретной общей реализации `ArrayList<Date>`. `DateList` является типом `ArrayList<Date>` и наследует отдельную реализацию всех методов, как он и наследовал бы от любого другого родителя. Мы даже можем назначить его обратно родительскому типу, если захотим, как показано в этом примере.

Общий подтип общего класса может либо расширять конкретную реализацию класса, как в прошлом примере, либо делить переменную типа, которая «передается» родителю при реализации:

```
class AdjustableTrap< T > extends Trap< T > {
public void setSize( int i ) { ... }
}
```

Здесь переменная типа `T`, использовавшаяся для реализации класса `AdjustableTrap`, передается для реализации базового класса `Trap`. Когда пользователь реализует `AdjustableTrap` для отдельного параметра типа, родительский класс также реализуется для этого типа.

Исключения и обобщения

Типы возникают в теле классов в другом месте — операторах методов `throws`. Мы можем использовать переменные типа для определения типа исключений, выбрасываемых методами, но чтобы это сделать, мы должны ввести понятие границ. Мы полнее изучим границы в следующей главе. В этом случае они применяются очень просто. Нам только нужно убедиться, что переменная типа, которую мы хотим использовать в качестве типа исключения, в действительности является типом `Throwable`. Мы можем сделать это, добавив оператор `extends` к объявлению нашей переменной типа таким образом:

```
< T extends Throwable >
```

Вот пример класса, параметризированного типом, который должен иметь тип `Throwable`.

Его метод `test()` принимает экземпляр этого типа объекта и выбрасывает его, как проверенное исключение:

```
ExceptionTester< T extends Throwable > {
public void test( T exception ) throws T {// выбрасываемый тип является
общим параметром
throw exception;
}
}
try {
new ExceptionTester<ClassNotFoundException>().test(
new ClassNotFoundException() );
} catch ( ClassNotFoundException e ) { ... }
```

Важной частью этого примера является то, что оператор `throws` нашего тестового метода определен для выбрасывания `T`, обобщенного типа параметра класса. Это значит, что мы можем параметризировать тип исключений, выбрасываемых классом.

Добавление границ накладывает ограничение, заключающееся в том, что тип параметра, используемый для реализации класса `T`, должен быть типом класса `Throwable`. И мы ссылались на тип `T` в операторе `throws`. Итак, `ExceptionTester<ClassNotFoundException>` может выбросить `ClassNotFoundException` из его метода `test()`. Отметьте, что это проверяемое исключение и тот факт, что оно не было потеряно компилятором. Компилятор приводит в жизнь проверенный тип исключения, который он только что применил.

Не обобщенные реализации `Throwable`

Мы видели, что переменная типа может использоваться, чтобы уточнить тип `Throwable` в операторе метода `throws`. По иронии судьбы, однако, мы не можем использовать обобщения для создания новых типов исключений. Никаких общих подтипов `Throwable` не разрешено. Если вы задумаетесь об этом на мгновение, вы поймете: для того, чтобы быть полезным, общий `Throwable` потребует блоков `try/catch`, которые могут дифференцировать реализации `Throwable`. И поскольку (опять же) нет реализаций обобщений во время выполнения, со стиранием это невозможно.

Ограничения, связанные с типом параметра

Мы видели, как типы параметра (переменные типа) общего класса используются для объявления переменных экземпляра, аргументов ме-

тода и возвращаемых значений, а также «передаются» для параметризации общих подклассов. Вопрос, о котором мы не говорили, это как мы можем использовать переменную типа общего класса или можем ли использовать ее вообще для создания экземпляров типа параметра или работы с объектами типа другими конкретными способами. Мы преднамеренно избегали этой проблемы в нашем прошлом примере «испытателя исключений», просто передавая объект исключения в качестве аргумента. Могли ли мы устранить этот аргумент? Ответ, к сожалению, заключается в том, что из-за ограничений стирания в действительности не существует информации типа параметра, с которой можно работать во время выполнения. В этой главе мы рассмотрим данную проблему и исследуем обходной путь.

Поскольку переменная типа `T` верно служила нам везде в качестве типа параметра, вам, наверное, представляется, что мы можем использовать ее для создания экземпляра `T`, вводя ключевое слово `new`. Но мы не можем:

```
T element = new T(); // Ошибка! Неверный синтаксис.
```

Запомните, что вся информация о типе стирается в скомпилированном классе. У необработанного типа нет никакой возможности узнать тип объекта, который вы хотите создать во время выполнения. Также нет никакого способа добраться до типа `Class` типа параметра через переменную типа по той же причине. Поэтому отражение также нам здесь не поможет. Это означает, что обобщения ограничены до работы с типами параметров в относительно автоматической манере (только по ссылке). Это одна из причин, по которой обобщения более полезны для контейнеров, чем для каких-либо других приложений. Эта проблема возникает часто, и у нее есть решение, хоть и не всегда такое простое, как нам бы хотелось.

Использование типа `Class<T>`

Единственный реальный способ получить информацию о типе, которая нам нужна во время выполнения, — это чтобы пользователь явно передал ссылку на тип `Class` в целом, как один из аргументов метода. Затем мы можем явно ссылаться к классу, используя отражение и создавая экземпляры или делать все, что необходимо. Это может казаться плохим решением, не обеспечивающим особой безопасности типов и возлагающим большой груз на разработчика, чтобы сделать все правильно. К счастью, мы можем использовать трюк с обобщениями, чтобы испол-

нить этот договор с пользователем и сделать его безопасным. Опять же, основная идея заключается в том, чтобы один из наших методов принял тип `Class` типа параметра, так что мы сможем использовать его во время выполнения. Продолжая наш «испытатель исключений»:

```
public void test( Class type ) throws T { ... }
```

Это не намного лучше, чем было раньше. В первую очередь, это не гарантирует, что тип `Class`, переданный методу, совпадет с параметризированным типом класса (здесь используется в операторе `throws`).

К счастью, сам класс `Class` теперь является обобщенным типом. Все экземпляры класса `Class`, созданные виртуальной машиной Java, реализуются их собственным типом в качестве параметра.

Класс типа `String`, например, теперь является `Class<String>`, а не просто каким-то произвольным экземпляром необработанного типа `Class`, который знает о строках.

Это имеет два последствия. Во-первых, мы не можем указать отдельную реализацию `Class`, используя тип параметра в нашем собственном классе. А во-вторых, поскольку класс `Class` является теперь общим, все методы отражения и создания экземпляра могут быть правильно типизированы и больше не нуждаются в приведении типов, поэтому мы можем написать наш метод `test()` таким образом:

```
public void test( Class<T> type ) throws T {
throw type.newInstance();
}
```

Единственный экземпляр `Class`, который может быть передан нашему методу `test()`, — это `Class<T>`, экземпляр `Class` для типа параметра `T`, на котором мы реализовали `ExceptionTester`. Итак, хотя на пользователе до сих пор лежит бремя передачи этого на вид чуждого аргумента `Class`, по крайней мере, компилятор гарантирует, что мы делаем это, и делаем безопасно:

```
ExceptionTester<ArithmeticException> et =
new ExceptionTester<ArithmeticException>();
et.test( ArithmeticException.class ); // никакой другой .class не
сработает
```

В этом отрывке кода попытка передать любой другой аргумент `Class` методу `test()` генерирует ошибку во время компиляции.

Границы

В процессе рассмотрения обобщений нам уже пришлось несколько раз упомянуть границы. Граница — это ограничение на тип типа параметра. Границы используют ключевое слово `extends` и некоторый новый синтаксис, чтобы ограничить типы параметров, которые могут быть применены к общему типу. В случае общего класса границы просто ограничивают тип, который может быть доставлен для его реализации.

Переменная типа может расширить класс или тип интерфейса, это означает, что его реализация должна быть этого типа или подтипа:

```
class EmployeeList< T extends Employee > { ... }
```

Здесь мы создали обобщенный тип `EmployeeList`, который может быть реализован только при помощи типов `Employee`. Далее мы можем потребовать, чтобы тип `Employee` реализовывал один или более интерфейсов, используя специальный синтаксис интерфейса `&`:

```
class EmployeeList< T extends Employee & Ranked & Printable > { ... }
```

Порядок границ интерфейса `&` незначителен, но только один тип класса может быть уточнен, а если он только один, он должен идти первым. Если у типа нет никаких особых границ, граница `extends Object` является косвенной.

Применяя границы к нашему типу, мы не только ограничиваем реализации обобщенного класса, но и делаем аргументы типа более полезными. Теперь, когда мы знаем, что наш тип должен расширять какой-то тип или применять набор интерфейсов, мы можем использовать переменные и аргументы, объявленные при помощи `T` теми другими типами имен. Вот немного натянутое расширение нашего предыдущего примера:

```
class EmployeeList< T extends Employee & Ranked & Printable >
{
Ranked ranking;
List<Printable> printList = new ArrayList<Printable>();
public void addEmployee( T employee ) {
this.ranking = employee; // T в качестве Ranked
printList.add( employee ); // T в качестве Printable
}
}
```

Этот пример показывает, что, накладывая границы на общий параметрический тип, мы можем требовать, чтобы он был отдельным типом класса или применял определенные типы интерфейса. Это позволяет нам использовать аргументы параметрического типа, переданного методам, более полезными способами. В этом примере мы знаем, что класс `EmployeeList` будет реализован общим типом, то есть типом `Printable`, и так мы можем использовать аргумент `employee` в качестве типа `Printable`.

Типизированные переменные могут также ссылаться на другие типизированные переменные в объявлении типа:

```
class Foo <A, B extends A> { ... }
```

Мы увидим частично неверный пример этого позже, когда будем говорить об определении класса `Enum`. Мы также увидим более удобную технику для объявления того, как индивидуальные элементы общего класса относятся к параметризованным типам, когда будем рассматривать знаки обобщения в следующем разделе.

Стирание и границы (работа со старым кодом)

Ранее в нашем разговоре о стирании мы упоминали, что результирующий тип, используемый вместо параметризованного типа в необработанном типе общего класса, является границей переменной типа. В частности, мы видели много обобщений без явных границ, которые сводились к границе типа `Object`. Мы также показали быстрый пример типа, который вводил границу `extends Date` и говорили, что типом его методов будет `Date`, а не `Object`. Теперь мы можем быть немного более точными.

Тип после использования стирания параметризованного типа общего класса является *крайней левой границей*; то есть первая граница, указанная после ключевого слова `extends` (буквально крайняя слева), становится типом, использованном в стирании.

Это предполагает, что если тип расширяет тип класса, он всегда является стираемым типом, потому что он всегда должен идти первым. Но если тип расширяет только типы интерфейса, выбор за нами. Этот тонкий вопрос важен для обратной совместимости с не общим кодом. Часто создавая общие версии не общих API, мы имеем возможность немного «подтягивать» спецификацию. Знание о крайней левой границе дает

нам способ явно контролировать тип стертого класса. Например, предположим, что мы создаем общий класс `List`, который хотим реализовывать только на объектах типа `Listable`, но мы предпочли бы не менять API нашего старого класса `List`, принимавшую элементы типа `Object`. Наша изначальная попытка

```
class List< E extends Listable > { ... }
```

производит необработанный тип, который принимает только тип `Listable`. Однако мы можем вставить немного необоснованный дополнительный тип, `Object`, в качестве крайней левой границы, чтобы получить назад нашу старую API, не меняя новые общие границы:

```
class List< E extends Object & Listable > { ... }
```

Вставка класса `Object` не меняет фактические границы общего класса, но меняет стертую сигнатуру.

Групповые символы

Ранее мы упоминали, что типы реализаций обобщенного типа, которые мы уже обсудили в этой главе, все были реализациями конкретного типа. Мы описывали это, подразумевая, что все аргументы параметров являются реальными типами Java. Например, `List<String>` и `List<Date>` являются реализациями общего класса `List` с конкретными типами `String` и `Date`. Теперь мы рассмотрим другой тип реализации обобщенного типа: *групповая реализация*.

Как мы увидим в этой главе, групповые символы являются для Java способом ввести полиморфизм в часть параметризованного типа общего равенства. Групповая реализация использует знак ? вместо действительного типа параметра во время реализации и означает, что тип может быть назначен любому диапазону возможных реализаций обобщенного типа. Сам групповой символ ? называется *безграничным групповым символом* и обозначает, что приемлем любой тип реализации (может быть назначен типу).

```
List<?> anyInstantiationOfList = new ArrayList<Date>();
anyInstantiationOfList = new ArrayList<String>(); // другая реализация
```

В этом отрывке мы объявили переменную `anyInstantiationOfList`, чей тип является реализацией с безграничным групповым символом

обобщенного типа List. (Как труднопроизносимо.) Это означает, что тип, который мы реализовали, может быть назначен любой отдельной конкретной реализации типа List, либо Dates, Strings, либо Foos. Здесь мы назначили ему сначала List<Date>, а в результате List<String>.

Супертип для всех реализаций

Реализация с безграничным групповым символом является подобием супертипа всех этих конкретных реализаций. В противовес виденным нами ранее отношениям обобщенного типа, которые придерживались только необработанных, «базовых» обобщенных типов, групповые обобщения позволяет нам реализовать полиморфизм для типов параметров. Неограниченный групповой тип является для параметров обобщенного типа тем же, что Object для обычных типов Java: супертипом всего.

```
// List<Object> не является List<Date>!
List<Object> objectList = new ArrayList<Date>() // Ошибка!
// List<?> может быть List<Date>
List<?> anyList = new ArrayList<Date>(); // Да!
```

В этом примере нам напоминают, что List<Object> не является List<Date>; полиморфизм не идет тем же путем, что и общие реализации конкретных типов. Но List<?>, неограниченная реализация группового типа, может быть назначена любой реализации List. Далее мы увидим, что групповые типы добавляют новое измерение передаче обобщенных типов.

Ограниченные групповые типы

Ограниченный групповой тип — это групповой тип, который использует ключевое слово extends как использовала бы переменная типа для ограничения диапазона назначаемых типов. Например:

```
List<? extends Date> dateInstantiations = new ArrayList<Date>();
dateInstantiations = new ArrayList<MyDate>(); // другая реализация
```

Наша переменная dateInstantiations ограничена до хранения реализаций параметризованных типов Date и их подклассов. Итак, мы можем назначить ей List<Date> или List<MyDate>. Так же как неограни-

ченный групповой тип служит суперклассом для всех реализаций обобщенного типа, ограниченные групповые типы создают более ограниченные супертипы, охватывающие более узкие масштабы реализаций. В этом случае наша реализация группового типа `List<?extends Date>` является супертипом всех реализаций `List` в типах `Date`. Так же как с границами параметризованного типа, граница `Date` называется общей границей типа.

Границы группового типа могут расширять интерфейсы так же, как использование типа & для добавления требований интерфейса к границе:

```
Trap< ? extends Catchable & Releaseable > trap;
```

В этом случае реализация служит супертипом набора реализаций для типов, реализующих интерфейсы `Catchable` и `Releaseable`.

Групповые типы в контексте контейнеров

Давайте разберемся в том, что означает групповой тип в контексте такого типа контейнера, как `List`. Реализация неограниченного типа может быть назначена любому типу реализации, но она в итоге относится к *реализации некоторого отдельного типа*. Реализация группового типа служит типом переменной, а переменная в итоге содержит некую фактическую *конкретную реализацию обобщенного типа*:

```
List<?> someInstantiationOfList;
someInstantiationOfList = new ArrayList<Date>();
someInstantiationOfList = new ArrayList<String>();
```

В этом примере наша переменная `List<?>` является либо `List <String>`, либо `List<Date>`. Это не какой-то новый тип `List`, который может содержать элементы `String` или `Date`.

Таким же образом групповой тип с границами в конечном счете содержит одну из конкретных реализаций, назначаемых его границам. Вообразите на одно мгновение, что у нас есть приватный класс `Foo` только с одним подклассом `Bar` и никакими другими. Выражение `Collection<? extendsFoo>` в этом случае означает набор двух возможностей: `Collection<Foo>` или `Collection<Bar>` — то есть либо элементы типа `Collection` с общим супертипом `Foo`, либо собрание элементов с общим супертипом `Bar`. Опять же, реализация обобщенного типа со-

впадает с каждой из этих реализаций обобщенного типа. Она не создает *новый* тип совокупности, который может содержать `Foo` или `Bar`. (Это в действительности задача `Collection<Foo>`, которая может содержать элементы `Foo` и `Bar`.)

По этой причине реализации группового типа являются действительными типами для отражения объекта, но они не могут использоваться в качестве типа для создания экземпляра объекта. В общем, вы не можете использовать групповой тип с ключевым словом `new`, чтобы назначить экземпляр объекта, поскольку групповой тип указывает на один объект или возможный набор объектов. Это не имеет смысла.

Нижние границы

Мы видели, как конструкция `extends` использовалась для указания верхней границы обоих переменных типа и реализаций группового типа. Она предполагает тип, который находится «на верхушке» иерархии объектов для границы. Реализации группового типа в действительности разрешают другой тип границ, называемый также *нижней границей*. Нижняя граница указывается ключевым словом `super` и, как вы можете догадаться, требует, чтобы реализации были определенного типа или любого его супертипа, до `Object`. Например:

```
List< ? super MyDate > listOfAssignableFromMyDate;
listOfAssignableFromMyDate = new ArrayList<MyDate>();
listOfAssignableFromMyDate = new ArrayList<Date>();
listOfAssignableFromMyDate = new ArrayList<Object>();
```

Эта реализация группового типа создает тип, который может содержать любую реализацию `List` типа `MyDate` или любого его супертипа. В мире нашего примера это означает, что групповой тип может быть назначен только одному из трех типов: `List<MyDate>`, `List<Date>` или `List<Object>`. Здесь мы обрезали иерархию наследования объекта после трех поколений. Никакие дальнейшие подклассы `MyDate` не могут быть использованы.

Как мы намекали в примере, это может помочь прочесть `? super MyDate` как «Передаваемый от `MyDate`». Нижние границы полезны для случаев, где мы хотим быть уверены, что отдельная реализация контейнера может хранить какой-то отдельный тип элемента, не ограничивая его до просто какого-то определенного типа элемента. Мы покажем хо-

роший пример, когда будем говорить об общих методах позже. Сейчас просто постарайтесь оценить это как дополнение к верхним границам. И последнее о нижних границах: только синтаксис групповых реализаций может использовать ключевое слово `super`, чтобы сослаться на нижние границы. Стирание замещает все ссылки к переменным типа их верхними границами, поэтому типы выполнения не имеют возможности осуществлять контракт.

Чтение, запись и арифметические операции

Мы пока обошли молчанием важную проблему в нашем обсуждении групповых типов: а именно, как мы можем их использовать? Какие виды типов применяет компилятор к переменным и аргументам, которые ссылаются на переменные типа в общем классе? Например, если у нас есть список `List<?>` любого типа реализации, каковы правила помещения объектов в его и изъятия из него? Каков их тип?

Нам необходимо взять два типа по отдельности. Исходя из аналогии с контейнером, мы вызовем получение возвращаемого значения из метода для объекта в качестве особого типа, *считывая объект как тип*. С другой стороны, мы вызовем передачу аргумента отдельного типа методам объекта, *записывая объект как тип*. Итак, например, `List<Date>` может быть прочитан и написан как тип `Date` и `Trap<Mouse>` имеет методы, которые могут быть прочтены и написаны как тип `Mouse`.

Хотя, чтобы быть более точными, мы должны сказать, что `List<Date>` может быть прочен как тип `Date`, но может быть написан как любой подтип `Date`. Все же мы можем добавить `MyDate` к `List<Date>`. Давайте посмотрим сейчас на реализацию `List< ? extends Date >`. Мы знаем, что она содержит реализацию типа `List` на каком-то типе `Date`. Что еще можем мы сказать о таких элементах, как `List`, которые могут содержать *любую* реализацию типа `Date`? Что ж, элементы всегда будут подтипами `Date`. Это означает, что как минимум мы должны быть способны прочесть объект через наш групповой тип как тип `Date`:

```
List< ? extends Date > someDateList = new ArrayList<MyDate>();
...
Date date = someDateList.get( 0 ); // читается как Date
```

Компилятор позволяет нам назначить значение непосредственно типу `Date`, поскольку он знает, что, какой бы ни была реализация `List`,

элементы должны быть подтипом Date. (Конечно, мы могли бы прочитать объект как тип Object или любой супертип Date, если бы хотели.)

Но насчет того, чтобы пойти другим путем, и насчет написания? Если бы someDatelist мог быть реализацией List на любом подтипе Date, как мы можем знать, какой тип объектов записывать туда? (Как мы можем безопасно вызвать метод add()?) Ответ — мы не можем. Поскольку мы не знаем правильный тип, компилятор не позволит нам написать ничего в List через нашу групповую реализацию типа:

```
List< ? extends Date > someDateList = new ArrayList<MyDate>();
someDatelist.add( new Date() ); // Ошибка во время компиляции!
someDatelist.add( new MyDate() ); // Ошибка во время компиляции!
```

Другими словами, это потому что наша групповая реализация имеет верхнюю границу Date, мы можем *считывать* тип как Date. Мы вскоре повторим это в форме правила.

Вспомните, что неограниченный групповой тип в действительности является только групповым типом с границей типа Object <? extends Object>. Очевидно, даже неограниченная реализация группового типа содержит объекты, которые могут быть назначены Object, поэтому нормально считывать неограниченный групповой тип в качестве типа Object:

```
List<?> someList = new ArrayList<String>();
...
Object object = someList.get( 0 ); // считывается как Object
```

Но, конечно, мы не можем знать настоящий тип элементов, поэтому не можем записывать в список через неограниченный групповой тип.

Что же насчет нижних границ? Что ж, ситуация искусно изменена с уважением к чтению и записи. Поскольку мы знаем, что элементы любой реализации, совпадающий с нашей ограниченным снизу групповым типом, должны быть супертипом нижней границы, мы можем осуществлять запись в объект как тип нижней границы через наш групповой тип:

```
List< ? super MyDate > listAssignableMyDate = new ArrayList<Date>();
listAssignableMyDate.add( new MyDate() );
listAssignableMyDate.add( new Date() ); // Ошибка во время компиляции!
```

Поскольку мы не знаем, к какому супертипу относятся элементы, мы не можем считывать список как какой-то отдельный тип. Конечно, класс `List` должен все еще содержать какой-то тип `Object`, поэтому мы можем всегда считывать список с нижней границей как тип `Object` через групповой тип. Тип `Object` является дефолтной верхней границей:

```
Object obj = listAssignableMyDate.get( 0 ); // считывается как Object
```

Что ж, мы изучили объяснение, и теперь можем просуммировать его сжато в правиле, которое легко запомнить:

> Реализации группового типа обобщенных типов могут быть прочтены как верхняя граница и записаны как нижняя граница.

Дополним: все реализации группового типа имеют верхнюю границу типа `Object`, даже если никакая другая не указана, поэтому все групповые реализации могут быть хотя бы прочтены как тип `Object`. Но не все групповые типы имеют нижнюю границу, и поэтому данные реализации группового типа могут быть записаны как тип более определенный, чем `Object`.

`<?>`, `<Object>` и необработанный тип

Мы изучили большой объем информации, и, может быть, немного тяжело понять семантику. Давайте потренируем наши знания, рассмотрев несколько случаев, которые могут иметь или не иметь сходства.

Естественные вопросы, которые возникают, это: какой же нам прок от неограниченных групповых типов? почему просто не использовать необработанные типы? как сравниваются реализации группового типа и необработанного типа? Первое отличие заключается в том, что компилятор выдает непроверенные предупреждения, когда мы используем методы необработанного типа. Но это поверхностно. Почему компилятор предупреждает нас? Потому что он не может уберечь нас от неправильного обращения с нашим необработанным типом и навязывания ему неверного типа объектов. Использовать неограниченный групповой тип — это как надеть боксерские перчатки и заявить, что мы хотим играть по правилам. Это может дорого обойтись. Компилятор гарантирует, что мы в безопасности, позволяя нам только те операции, которые он считает безопасными, а именно чтение в качестве типа `Object` (верхняя граница всего). Компилятор вообще не позволяет нам делать запись в неограниченный групповой тип. Поэтому зачем использовать неограниченный групповой тип? Чтобы играть по правилам обобщений и гарантировать, что мы не делаем ничего небезопасного.

Далее мы можем опровергнуть мнение о том, что реализация неограниченного группового типа похожа на любую реализацию типа `Object`. Помните, что `List<?>` содержит *некую* реализацию `List`. Насколько нам известно, это может быть `List<Date>`. Но `List<Object>` в действительности является списком, который содержит конкретные типы `Object`. `List<Object>` может считываться и записываться как `Object`. `List<?>` может только считываться (не записываться) и только считываться как тип `Object` в дегенерированном смысле. Все элементы `List<?>` в действительности являются неким неизвестным типом. Элементы списка неизвестного типа все имеют общий супертип, который может быть типом `Object` или каким-то другим общим типом, который является более ограничивающим, чем `Object`. Знание того, что «могло бы быть» в `List<?>`, на практике мало что нам дает, но означает что-то совсем отличное от `List<Object>`.

В конце концов, давайте завершим сравнения, спросив, как сравниваются `List<Object>` и необработанный тип. Теперь мы подошли близко к чему-то. В действительности необработанный тип после стирания в сущности окажется `List<Object>`, когда вы его отзовете. Но в этом случае мы говорим компилятору, что это нормально. Здесь мы просим тип с элементами, которые могут содержать любую безопасность типа и обязательства компилятора. Ответ на вопрос, как сравниваются `List<Object>` и необработанный тип `List`, заключается в том, что `List<Object>` является «общебезопасной» версией необработанного типа вчерашнего дня.

Отношения группового типа

Прежде чем мы закончим наше обсуждение групповых типов, давайте вернемся еще раз к понятию реализаций группового типа в качестве типов в системе типов Java. Ранее в этой главе мы описали, как обычные конкретные реализации обобщенных типов связаны в силу их «базового» наследования обобщенного типа, только с оговоркой, что их параметры типов в точности совпадают. Позже мы пытались ввести идею, что реализации группового типа добавляют отношение наследования параметрам типа, что является второй половиной общей реализации. Теперь мы объединим две половины. Очень скоро все может стать загадочным, но простые случаи усвоить легко.

Вопрос в следующем: если мы имеем две разные групповые реализации типа или связанных типов, то как они связаны, если связаны во-

обще? Например, может ли неограниченному групповому типу быть назначено значение с более ограничивающей границей, поскольку оно может содержать любую реализацию?

```
List< ? extends Date > dateLists = ...;
List< ? > anylists;
anyLists = dateLists; // Ok!
```

Ответ: да. С целью присвоения групповые реализации могут рассматриваться как типы с возможными супертипами или отношениями супертипа, определенными их границами. Давайте разберем реализацию неограниченного типа такой, какой она действительно является, реализацией с верхней границей `Object`:

```
List< ? extends Date > dateLists = ...;
List< ? extends Object > objectLists;
objectLists = dateLists; // Ok!
```

Правило заключается в том, что если «базовый» общий, необработанный тип является присваиваемым и границы групповой реализации также присваиваемы, всеобщий тип также является присваиваемым. Давайте посмотрим на другой пример:

```
List< ? extends Integer > intLists = ...;
Collection< ? extends Number > numCollections;
numCollections = intLists; // Ok!
```

Это, в сущности, говорит о том, что некоторые типы `List` или `Integer` могут рассматриваться как типы `Collection` типов `Number` через реализацию группового типа. Если вы обдумаете это, то увидите, что никакого противоречия здесь нет. `List` определенно является интерфейсом `Collection`. И все, что мы делаем, — это расширяем тип, при помощи которого можем считать элементы от `Integer` до `Number`. Однако ни в одном случае мы не можем сделать запись в коллекцию через реализацию группового типа.

В итоге это означает, что с введением групповых реализаций отношения типов в обобщенных классах Java становятся двумерными. Следует рассмотреть отношения необработанного типа, а затем отношения группового параметра. В действительности, если вы учтете, что обобщенные классы могут иметь более одного типа параметров, отношения могут стать даже более сложными (*N*-мерными). К счастью, все это нечасто встречается в реальном мире.

Обобщенные методы

До сих пор в этой главе мы говорили об обобщенных типах и реализации обобщенных классов. Теперь мы собираемся рассмотреть другой тип: *обобщенные методы*. Фактически обобщенные методы делают для индивидуальных методов то, что параметры типов делают для обобщенных классов. Но, как мы увидим, обобщенные методы умнее и могут узнать свои типы параметров из контекста своего применения без необходимости быть явно параметризованными. (В действительности, конечно, это делает компилятор.) Обобщенные методы могут возникнуть в любом классе (не только в обобщенных классах) и полезны для множества различных приложений.

Во-первых, давайте еще раз быстро посмотрим, как обычные методы взаимодействуют с обобщенными типами. Мы видели, что обобщенные классы могут содержать методы, которые используют переменные типа в своих аргументах и возвращаемых типах, чтобы адаптироваться к параметризации класса. Мы также упоминали, что сами обобщенные типы могут использоваться в большинстве случаев, когда могут быть использованы другие типы. Поэтому методы обобщенных или не обычных классов могут также использовать обобщенные типы в качестве аргументов и возвратных типов. Вот примеры такого использования:

```
// Необобщенные методы
class GenericClass< T > {
// метод, использующий тип обобщенного параметра класса
public void T cache( T entry ) { ... }
}
class RegularClass {
// метод, использующий обобщенный тип
public List<Date> sortDates( List<Date> dates ) { ... }
// метод, использующий групповой обобщенный тип
public List<?> reverse( List<?> dates ) { ... }
}
```

Метод `cache()` в `GenericClass` принимает аргумент типа параметра `T`, а также возвращает значение типа `T`. Метод `sortDates()`, который появляется в не общем классе примера, работает с конкретным обобщенным типом, а метод `reverse()` работает с групповой реализацией обобщенного типа. Это примеры методов, которые работают с обобщениями, но это не настоящие обобщенные методы.

Знакомство с обобщенными методами

Как обобщенные классы, обобщенные методы имеют объявление типа параметра с использованием синтаксиса <>. Этот синтаксис появляется перед возвращаемым типом метода:

```
// обобщенный метод
<T> T cache( T entry ) { ... }
```

Этот метод `cache()` очень похож на предыдущий пример, кроме того, что он имеет собственное объявление типа параметра, которое определяет переменную типа `T`. Этот метод является обобщенным методом и может возникнуть в общем и не общем классах. Масштаб `T` ограничен до метода `cache()` и скрывает любое определение `T` в любом включающем общем классе. Как и обобщенные классы, тип `T` может иметь границы:

```
<T extends Entry & Cacheable > T cache( T entry ) { ... }
```

В отличие от общего класса, он не должен быть реализован отдельным типом параметра для `T` до того, как он используется. Вместо этого, он *выводит* параметр `T` из типа его аргумента, `entry`. Например:

```
BlogEntry newBlogEntry = ...;
NewspaperEntry newNewspaperEntry = ...;
BlogEntry oldEntry = cache( newBlogEntry );
NewspaperEntry old = cache( newNewspaperEntry );
```

Здесь наш обобщенный метод `cache()` вывел тип `BlogEntry` (который, как мы предположим ради примера, является типом `Entry` и `Cacheable`). `BlogEntry` стал типом `T` возвращаемого типа, мог быть использован в любом типе `Entry` и был способен вернуть новый тип точно таким же образом. Вот что является сильной стороной обобщенных методов: возможность выводить параметризованный тип из контекста его использования. Мы углубимся в детали этого позже. Другое отличие компонентов общего класса заключается в том, что обобщенные методы могут быть статическими:

```
class MathUtils {
public static <T extends Number> T max( T x, T y ) { ... }
}
```

Конструкторами классов фактически являются также методы, и они следуют тем же правилам, что и обобщенные методы, кроме возвращаемого типа.

Выведение типа из аргументов

В предыдущей главе мы видели, как метод выводил свой тип из аргумента:

```
<T> T cache( T entry ) { ... }
```

Но что если аргументов больше чем один? Мы видели именно эту ситуацию в предыдущем отрывке, статический обобщенный метод `max( x, y )`. Все выглядит хорошо, когда мы даем ему два идентичных типа:

```
Integer max = MathUtils.max( new Integer(1), new Integer( 2 ) ) ;
```

Но что он делает с аргументами в этом вызове?

```
MathUtils.max( new Integer(1), new Float( 2 ) ) ;
```

В этом случае компилятор Java поступает действительно умно. Он взбирается по типам аргументов родительских классов, ища *ближайший общий супертип*. Java также идентифицирует *ближайшие реализованные общие интерфейсы* обоих типов. Он определяет, что типы `Integer` и `Float` являются подтипами типа `Number`. Он также распознает, что каждый из них реализует (определенную общую реализацию) интерфейса `Comparable`. Java затем эффективно делает объединение типов параметризованным типом `T` для вызова этого метода. Результирующим типом, если использовать синтаксис ограничений, является `Number & Comparable`. Это означает для нас, что результирующий тип `T` назначаем всему, что совпадает с этой отдельной комбинацией типов.

```
Number max = MathUtils.max( new Integer(1), new Float( 2 ) );
Comparable max = MathUtils.max( new Integer(1), new Float( 2 ) );
```

В переводе на русский, это утверждение означает, что мы можем работать с нашими типами `Integer` и `Float` в одно и то же время, только если мы думаем о них, как о типах `Number` или `Comparable`, что имеет смысл. Возвращаемый тип стал новым типом, который фактически является `Number`, что также реализует интерфейс `Comparable`.

Эта логика интерфейса работает с любым количеством аргументов. Но чтобы быть полезными, аргументы в действительности должны делить какой-то важный общий супертип или интерфейс. Если у них нет ничего общего, результатом будет их фактический общий прародитель, тип `Object`. Например, ближайший общий супертип `String` и `List` — это

`Object` вместо с интерфейсом `Serializeable`. Метод не может сделать многое с типом, которому не хватает реальных границ.

Выведение типа из контекста назначений

Мы видели, как обобщенный метод выводит свой параметризованный тип из его типов аргументов. Но что если переменная не используется ни в одном аргументе метода или метод не имеет аргументов? Предположим, что у метода есть только параметрический возвращаемый тип:

```
<T> T foo() { ... }
```

Вы можете подумать, что это ошибка, поскольку окажется, что у компилятора нет возможности определить, какой тип мы хотим. Но это не так! Компилятор Java достаточно умен, чтобы посмотреть на контекст, в котором вызывается метод. Особенно если результат метода назначен переменной, компилятор пытается сделать из типа той переменной параметризованный тип. Вот пример. Мы создадим фабрику для наших объектов `Trap`:

```
<T> Trap<T> makeTrap() { return new Trap<T>(); }
// использование
Trap<Mouse> mouseTrap = makeTrap();
Trap<Bear> bearTrap = makeTrap();
```

Как будто по волшебству компилятор определил, какой тип реализации `Trap` мы хотим, основываясь на контексте назначений.

Но не торопитесь ликовать по поводу открывающихся возможностей: вы не так уж много сможете сделать с простым параметром типа в теле этого метода. Например, мы не можем создать экземпляры никакого отдельного конкретного типа `T`, что ограничивает полезность фабрик. Почти все, что мы можем сделать, — это то, что мы показываем здесь, создавая экземпляры обобщений, параметризованных верно для контекста.

Более того, интерфейс работает только на назначении переменной. Java не пытается угадать тип параметра, основанного на контексте, если вызов метода используется другими способами, такими как создание аргумента для метода или значение оператора возврата метода. В этих случаях созданный тип возвращается к типу `Object`. (См. раздел «Явный вызов типа» для пояснений.)

Явный вызов типа

Хотя это и не понадобится вам часто, существует синтаксис для вызова обобщенного метода с особыми параметризованными типами. Синтаксис немного громоздкий и включает префикс объекта класса или интерфейса, за которым следует знакомый список типа в угловых скобках, поставленный перед фактическим вызовом метода. Вот некоторые примеры:

```
Integer i = MathUtilities.<Integer>max( 42, 42 );
String s = fooObject.<String>foo( "foo" );
String s = this.<String>foo( "foo" );
```

Префикс должен быть экземпляром класса или объекта, содержащим метод. Одна из ситуаций, где вам может понадобиться использовать явный вызов типа, это если вы вызываете обобщенный метод, который выводит свой тип из контекста назначения, но не назначаете значение переменной напрямую. Например, если бы вы хотели передать результат нашего метода `makeTrap()` в качестве параметра другому методу, в обратном случае он вернется к `Object`.

Перехват группового типа

Общие методы могут сделать для нас еще один трюк, включающий смягченные реализации группового типа обобщенных типов. Понятие *перехват группового типа* относится к факту, что обобщенные методы могут работать с аргументами, чей тип является групповой реализацией типа, как если бы они были известны:

```
<T> Set<T> listToSet( List<T> list ) {
Set<T> set = new HashSet<T>();
set.addAll( list );
return set;
}
// использование
List<?> list = new ArrayList<Date>();
Set<?> set = listToSet( list );
```

Результат этих примеров заключается в том, что мы превратили неизвестную реализацию `List` в неизвестную реализацию `Set`. Переменная типа `T` представляет действительный тип аргумента, `list`, для целей тела метода. Групповая реализация должна совпадать с любыми

границами метода параметризованного типа. Но поскольку мы можем работать с переменной типа только через типы границ, компилятор позволяет спокойно ссылаться к ней по этому новому имени, `T`, как если бы это был известный тип. Это может казаться не очень интересным, но это полезно, поскольку позволяет методам, которые принимают групповые реализации типов, делегировать свою работу другим общим методам. Если взглянуть на это иначе, обобщенные методы являются более мощной альтернативой методам, использующим групповые реализации типов. Далее мы проведем небольшое сравнение.

Выбор между групповыми типами и обобщенными методами

Вы вспомните, что попытка работать с объектом через групповую реализацию его обобщенного типа ограничивает нас до «чтения» объекта. Мы не можем «записывать» типы в объект, поскольку его тип параметра неизвестен. Напротив, поскольку обобщенные методы могут вводить или «захватывать» фактический тип их аргументов, они позволяют нам делать намного больше с широким диапазоном типов, чем мы могли с одними реализациями группового типа.

Например, предположим, что мы хотим записать вспомогательный метод, который заменяет первые два элемента списка. Используя групповой тип, мы бы написали что-то наподобие этого:

```
// Плохая реализация
List<?> swap( List<?> list ) {
Object tmp = list.get(0);
list.set( 0, list.get(1) ); // ошибка, невозможно записать
list.set( 1, tmp ); // ошибка, невозможно записать
return list;
}
```

Но нам не разрешается вызывать метод `set()` для нашего списка, поскольку мы не знаем, какой тип он в действительности содержит. Мы застряли и сможем сделать не так много. Но соответствующий обобщенный метод дает нам реальный тип, с которым можно иметь дело:

```
<T> List<T> swapGeneric( List<T> list ) {
T tmp = list.get( 0 );
list.set( 0, list.get(1) );
```

```
list.set( 1, tmp );
return list;
}
```

В наших силах объявить здесь переменную конкретного (выведенного) типа и сделать запись, используя верно методы `set()`. Может показаться, что обобщенные методы — это наилучший выбор здесь. Но есть и третий путь. Групповой перехват, как описано в предыдущей главе, позволяет нам делегировать нашу версию группового типа метода нашему фактическому обобщенному методу и использовать его как если бы тип был введен, пусть даже он и открытый:

```
List<?> swap( List<?> list ) {
return swapGeneric( list ); // делегируется общей форме
}
```

Здесь мы делегировались общей версии.

Массивы параметризованных типов

Есть еще одно место, где мы не рассматривали, как обобщенные типы влияют на язык Java: типы массивов. После всего увиденного нами будет естественно ожидать, что массивы обобщенных типов примут участие в деле. Но, как мы увидим, у Java противоречивые отношения с массивами параметризованных типов.

Первое, что нам нужно сделать, это вспомнить, как работают массивы с обычными типами Java. Массив — это что-то наподобие встроенной библиотеки некоего базового типа элемента. Более того, типы массивов (включая все многомерные варианты массивов) являются настоящими типами в языке Java и представляются во время выполнения уникальными типами класса. Вот где начинаются неприятности. Хотя массивы в Java ведут себя во многом как общие библиотеки (они изменяют свои API, чтобы адаптироваться к определенному типу «чтения» и «записи»), они не ведут себя как обобщения Java в том, что касается их типовых отношений. Как мы видели в главе 6, массивы существуют в иерархии классов Java, возникая от `Object` и распространяясь по параллельным ветвям простых объектов Java.

Массивы являются *ковариантными подтипами* других типов массивов, это значит, что в отличие от конкретных обобщенных типов, хоть

они и меняют их сигнатуры методов, они все же относятся к своим родителям. Это означает, что `Strings []` в Java является подтипом `Object []`. Это поднимает проблему псевдонимов, которую мы упоминали ранее. Массив типов `String` может быть использован под псевдонимом массивов типа `Object`, и мы можем попытаться поместить элементы в него нелегально, что не будет замечено до момента выполнения:

```
String [] strings = new String[5];
Object [] objects = strings;
objects[0] = new Date(); // Исключение во время выполнения
ArrayStoreException!
```

Чтобы предотвратить катастрофу, Java должен проверять каждое назначение массива на правильность типа во время выполнения. Но вспомните, что обобщенные типы не имеют действительных представлений во время выполнения; есть только необработанный тип. Поэтому у Java не будет возможности узнать разницу между элементами `Trap<Mouse>` и `Trap<Bear>` в массиве, если был применен псевдоним, например, такой как `Object []`. По этой причине Java не позволяет вам создавать массивы обобщенных типов — по крайней мере, конкретных. (Подробнее об этом мы расскажем далее в этой главе.)

Использование типов массивов

После того как мы сказали, что Java не позволит вам создавать эти массивы, вы наверняка думаете, что это, пожалуй, конец истории. Но нет! Хоть даже у нас и нет настоящих реализаций, которые выполняют необходимое поведение во время выполнения, Java в любом случае позволяет нам объявлять тип массива. Загвоздка в том, что вам придется нарушить безопасность типов, чтобы использовать их, применив массив необработанного типа в качестве их реализации:

```
Trap<Mouse> [] tma = new Trap[10]; // непроверенное предупреждение
Trap<Mouse> tm = new Trap<Mouse>();
tma[0] = tm;
Trap<Mouse> again = tma[0];
```

Здесь мы объявили массив обобщенного типа `Trap<Mouse>`. Назначение любого значения (отличного от `null`) этой переменной, `tma`, будет иметь результатом непроверенное предупреждение компилятора во время назначения.

Что мы по сути говорим здесь компилятору — это просим доверять нам в том, что массив содержит только правильные обобщенные типы, и просим его разрешить нам использовать его впоследствии, как будто он проверен. Мы не получаем предупреждения при каждом использовании, как получали бы при использовании необработанного типа, а только там, где мы назначаем массив. Загвоздка в том, что компилятор не может уберечь нас от неправильной эксплуатации массива. Непроверенное предупреждение в том месте, где мы назначаем массив, — это просто типичное предупреждение, которое напоминает нам, что можно неправильно эксплуатировать массив.

Чем хороши массивы обобщенных типов?

Почему Java вообще позволяет нам объявлять массивы обобщенных типов? Одна из важных причин: это позволяет использовать обобщенные типы в методах аргументов переменной длины. Например:

```
void useLists( List<String> ... lists ) {
List<String> ls0 = lists[0];
}
```

Другой ответ заключается в том, что это удобный предлог сохранить возможность использовать массивы при необходимости. Вы можете захотеть сделать это по крайней мере по двум причинам. Во-первых, массивы во многих случаях быстрее библиотек. Время выполнения Java очень хорошо в оптимизации доступа к массиву, и иногда можно просто принять предупреждение компилятора, потому что это стоит того. Во-вторых, есть проблема связывания с помощью интерфейса общего кода и унаследованного кода, и тут только Javadoc и ваша вера в разработчика являются гарантиями содержания. При назначении массивов необработанного типа общим реализациям мы можем хотя бы обеспечить, что при простом использовании мы не злоупотребляем типами в новом коде.

Групповые типы в типах массивов

В общем групповые реализации обобщений могут использоваться в качестве базового типа массивов таким же образом, как конкретные реализации. Давайте посмотрим на пример:

```
ArrayList<?>[] arrayOfArrayLists = ...;
```

Это объявление типа является массивом неограниченных групповых реализаций типа `ArrayList`. Каждый элемент массива может содержать экземпляр группового типа, в данном случае это означает, что каждый элемент массива может содержать отдельную реализацию `ArrayList`. Например:

```
arrayOfArrayLists[0] = new ArrayList<Date>();
arrayOfArrayLists[1] = new ArrayList<String>();
```

Также существует сюрприз, которым мы собираемся вас удивить в отношении групповых типов в массивах. Хотя мы сказали, что Java не позволяет нам создавать массивы обобщенных типов, существует исключение из этого правила. Java позволяет нам создавать массивы неограниченных реализаций группового типа. Вот два примера:

```
ArrayList<?>[] arrayOfArrayLists = new ArrayList<?>[10];
arrayOfArrayLists[0] = new ArrayList<Date>();
Trap<?> [] arrayOfTraps = new Trap<?>[10];
arrayOfTraps[0] = new Trap<Mouse>();
```

Здесь мы не только объявили два массива реализаций группового типа, но и назначили массивы! Хитрость заключается в том, что массивы должны быть неограниченного группового типа. Почему это работает? Потому что каждый элемент массива в неограниченной групповой реализации массива может содержать любую реализацию, в проверке общей части во время выполнения нет необходимости. Любая реализация является назначаемой элементу типа `ArrayList<?>`, поэтому требуется только проверка необработанного типа.

Понятие *материализуемого типа* используется, чтобы сослаться на любой объект, который не меняется при стирании. Это включает простые конкретные типы Java, простые и неограниченные групповые реализации. Материализуемые типы являются подобием реальных типов в «Матрице»: они все еще существуют после отключения от симуляции.

Пример: класс `Enum`

Если вы посмотрите на класс `java.lang.Enum` в Java 5 или более поздней версии, то увидите довольно странно выглядящее объявление обобщенного типа:

```
Enum< E extends Enum<E> > { ... }
```

В попытке разобраться с этим вам могут помешать две мысли, которые мы постараемся сейчас же разогнать. Во-первых, это может оказаться рекурсивным. Кажется, что переменная типа E определена как что-то, что еще не определено до конца. Но в действительности это не так. Часто мы сталкиваемся с математическими уравнениями в виде `x = function( x )`, и они не являются рекурсивными. Чего они действительно требуют, так это особого значения `x`, которое удовлетворяет условию. Далее, хотя довольно ясно, что E является подтипом некой формулировки обобщенного типа `Enum`, вы сможете сделать поспешный вывод, что сам E должен быть общим типом. Запомните, что конкретные типы могут расширять обобщения так же, как это могут делать обобщения. Держа это в уме, давайте искать какую-то классификацию, которая будет соответствовать этим границам. На секунду сфокусируемся только на границе:

```
E extends Enum<E>
```

`E` является подклассом некоторой параметризации `Enum` и, в частности, параметризация `Enum` на самом типе подкласса. Повторим, он требует, чтобы любые реализации типа `Enum` были подклассами некоторых параметров типа `Enum`. И более того, параметризации типа `Enum` подают их собственный тип как типовой параметр их родителя, `Enum`. Какой тип класса удовлетворяет этому условию?

```
class Foo extends Enum<Foo> { }
```

Этот класс `Foo` удовлетворяет. Объявление `Foo` в действительности считывает так же, как граница. `Foo` — это простой конкретный тип, который расширяет `Enum`, параметризованный его собственным типом*.

Что конкретно это выполняет? Первое значение этой классификации — это то, что `Enum` должен быть реализован дочерним типом в качестве его параметризованного типа. Это означает, что любые методы родительского класса `Enum`, который ссылается к переменной типа E, будут теперь ссылаться к *дочернему* типу. Эта особая граница гарантировала, что дочерние типы настраивают своего родителя собственным типом. В действительности, это в точности то, в чем нуждается класс `Enum` в Java, чтобы работали исчисления. Метод `compareTo()` в исчислении Java ссылается на переменную типа и предназначен для того, чтобы быть применимым только к другим реализациям отдельного родительского типа `enum`:

---

* В реальной жизни Java не позволяет нам расширять тип `Enum`, это предназначено для ключевого слова `enum` и компилятора. Но структура такая же, как показана.

```
public int compareTo( E e ) { ... }
```

Например, исчисляемый тип `Dog` должен быть способен сравнивать только типы `Dog`, и сравнение `Dog` с `Cat` должно привести к ошибке во время компиляции. Граница выполняет только это, адаптируя метод `compareTo()` под тип `Dog`:

```
class Dog extends Enum<Dog> { ... }
```

Обычно не окончательный базовый класс, который не имеет возможности узнать, какие дети у него появятся в будущем, может только сослаться на собственный тип как на общий супертип всех детей, когда он хочет работать с другими типами своего вида. Методы не общего класса `Enum` могут только снабжать методы, которые работают с любым классом `Enum`. Но через магию обобщений мы можем эффективно менять интерфейс API класса, основанный на том, как он вызывается через параметр общего класса, приспосабливая эти методы под себя и отправляя базовый тип на целое поколение.

Пример: метод `sort()`

Копаясь в классе `java.util.Collections`, мы находим все типы методов статических утилит для работы с библиотеками. Среди них и эта конфетка — статический обобщенный метод `sort()`:

```
<T extends Comparable<? super T>> void sort( List<T> list ) { ... }
```

Еще один орешек, который нужно расколоть. Давайте сфокусируемся на последней части границы:

```
Comparable<? super T>
```

Это групповая реализация интерфейса `Comparable`, поэтому мы можем читать `extends` как `implements`, если это поможет. `Comparable` содержит метод `compareTo()` для некоторого типа параметра. `Comparable<String>` означает, что метод `compareTo()` принимает тип `String`. Вследствие этого `Comparable<? super T>` является набором реализаций на `T` и всех его суперклассах. `Comparable<T>` удовлетворяет, и то же касается `Comparable<Object>`. Это означает, что элементы должны быть сравнимы со своим собственным типом или неким супертипом их собственного типа. Этого достаточно, чтобы гарантировать, что элементы могут сравниваться друг с другом, но это не так ограничивает, как утверждение,

что они сами должны реализовывать метод `compareTo()`. Некоторые элементы могут наследовать интерфейс `Comparable` от родительского класса, который знает, как сравнивать только супертип `T`, и это как раз то, что здесь разрешено.

Заключение

Обобщения Java являются очень мощными и полезными добавлениями к языку. Хотя некоторые детали, в которые мы погрузились ближе к концу этой главы, могут показаться пугающими, в целом все очень просто и убедительно: обобщения делают библиотеки лучше. Когда вы начнете писать больше кода, используя обобщения, то обнаружите, что ваш код становится лучше читаемым и более понятным. Обобщения делают явным то, о чем ранее приходилось догадываться при работе с языком.

Глава 9
ПОТОКИ ВЫПОЛНЕНИЯ

Мы принимаем как должное, что современные компьютерные системы могут управлять работой многих приложений и операционных систем (ОС) параллельно и делают возможным то, что все программное обеспечение работает одновременно. Несмотря на то что большинство систем сегодня имеют множество процессоров и некоторые процессоры способны на ухищрения, обеспечивающие некоторую степень параллелизма, по большей части процессор в действительности может справляться только с одной задачей за один раз, и то, что мы видим, — это ловкость рук операционной системы, которая жонглирует приложениями и переключает свое внимание между ними так быстро, что кажется, будто они работают одновременно.

В старые времена единицей параллелизма для таких систем служило приложение, или *процесс*. Для операционной системы процесс был в той или иной степени черным ящиком, который сам решал, что делать. Если приложение требовало большего параллелизма, оно могло получить его, только запустив множество процессов и обмениваясь данными с ними, но это был тяжеловесный и не очень изящный подход. Позже было введено понятие *потоков*. Потоки обеспечивают мелкомодульный параллелизм внутри процесса под контролем самого приложения. Потоки существовали долгое время, но исторически их было тяжело использовать. В Java поддержка потоков встроена в сам язык, что делает работу с ними легче. Утилиты параллелизма Java обращаются к общим шаблонам и практикам в многопоточных приложениях и поднимают их до уровня реальных API Java. В совокупности это означает, что Java — это язык, который поддерживает потоки исходно и на высоком уровне. Это также означает, что Java извлекает максимальную выгоду из потоков, поэтому важно, чтобы вы с самого начала приступили к знакомству с ними. Не всем разработчикам понадобится написать приложения, которые явно используют потоки или параллелизм, но многие используют некоторые связанные с ними возможности.

Потоки выполнения являются составной частью дизайна многих API Java, особенно тех, что вовлечены в приложения клиентской сторо-

ны, графику и звук. Например, когда мы рассмотрим программирование GUI позже в этой книге, мы увидим, что метод компонента `paint()` не вызывается непосредственно приложением, а скорее отдельным рисующим потоком в системе выполнения Java. В любой момент времени многие такие фоновые потоки могут выполнять действия параллельно с вашим приложением. На стороне сервера пишущий код, который явно осуществляет обработку потока, является менее распространенным и активно предотвращается в контексте серверов приложений и веб-приложений. В этих сценариях серверная среда должна контролировать распределение времени. Однако потоки Java находятся там, обслуживая каждый запрос и запуская компоненты вашего приложения. Важно понять, как ваш код подходит этой среде.

В этой главе мы поговорим о написании приложений, которые создают и используют свои потоки явно. Мы поговорим о низкоуровневой поддержке потоков, изначально встроенной в язык Java, а затем обсудим пакет утилит потока `java.util.concurrent` в подробностях в конце этой главы.

Введение в потоки

Концептуально *поток* — это поток управления в программе. Поток подобен более знакомому понятию *процесса*, кроме того, что потоки в том же приложении намного больше связаны и в основном делят одно и то же состояние. Это похоже на поле для гольфа, на котором играют много гольфистов одновременно. Потоки взаимодействуют, чтобы делить рабочее пространство. Они имеют доступ к одним и тем же объектам, включая статические переменные и переменные экземпляра в их приложении. Однако потоки имеют собственные копии локальных переменных, так же как игроки делят поле для гольфа, но не делят некоторые личные вещи, такие как клюшки и шары. У множества потоков в приложении те же проблемы, что и у гольфистов, — одним словом, синхронизация. Как две группы игроков не могут вслепую играть на одной зеленой лужайке в одно и то же время, у вас не может быть несколько потоков, пытающихся получить доступ к тем же переменным без подобия согласования. Кто-то может пострадать. Поток может заслужить право использовать объект, пока не закончил со своим заданием, как команда, играющая в гольф, получает эксклюзивные права на лужайку, пока не закончит. И поток, который является более важным, может поднять свой приоритет, утвердив свое право на игру. Дьявол скрывается

в деталях, конечно, и эти детали исторически сделали потоки сложными в употреблении. К счастью, Java делает создание, управление и согласование потоков проще, интегрировав некоторые из этих концепций непосредственно в язык. Нередко люди застревают на потоках, начиная работать с ними, поскольку создание потока тренирует многие ваши новые навыки по Java одновременно. Вы можете избежать путаницы, запомнив, что в запуск пакета всегда вовлечены два участника: объект `Thread` языка Java, который представляет сам поток, и произвольно нацеленный объект, содержащий метод, который должен выполнить поток. Позже вы увидите, что можно при помощи простой ловкости рук объединить эти две роли, но этот особый случай просто меняет компоновку, а не отношения.

Класс `Thread` Class и интерфейс `Runnable`

Новый поток рождается, когда мы создаем экземпляр класса `java.lang.Thread`. Объект `Thread` представляет реальный поток в интерпретаторе Java и служит рычагом контроля и координации его выполнения. С его помощью мы можем создать поток, подождать, пока он выполнится, заставить его заснуть на какое-то время или прервать его активность. Конструктор класса `Thread` принимает информацию о том, где поток должен начать выполнение. В принципе мы просто хотели бы сказать ему, какой метод запускать, но поскольку в Java нет указателей на метод (по крайней мере не в таком смысле), мы не можем указать ему метод прямо. Вместо этого мы должны зайти с другой стороны и использовать интерфейс `java.lang.Runnable`, чтобы создать или отметить объект, который содержит «выполняемый» метод. `Runnable` определяет единственный метод `run()` широкого применения:

```
public interface Runnable {
abstract public void run();
}
```

Каждый поток начинает свою жизнь, выполняя метод `run()` в объекте класса `Runnable`, который является «целевым объектом», переданным конструктору потока. Метод `run()` может содержать любой код, но он должен быть публичным, не принимать никаких аргументов, не иметь возвращаемого значения и не выбрасывать проверяемых исключений. Любой класс, который содержит подходящий метод `run()`, может заявить, что он реализует интерфейс `Runnable`. Экземпляр этого класса тогда является выполняемым объектом, который может быть целью но-

вого потока. Если вы не хотите помещать метод `run()` непосредственно в ваш объект (и очень часто вы не хотите), вы всегда можете создать класс-адаптер, который будет служить для вас интерфейсом `Runnable`. Метод адаптера `run()` затем может вызвать любой метод после того, как поток начат. Мы покажем примеры этих опций позже.

Создание и запуск потоков

Новорожденный поток остается неактивным, пока мы не дадим ему символический шлепок по мягкому месту, вызвав метод `start()`. Затем поток просыпается и продолжает выполнять метод `run()` своего целевого объекта. `start()` может быть вызван только однажды во время существования потока. Когда поток запущен, он продолжает выполняться, пока метод `run()` целевого объекта не вернется (или не выбросит непроверяемое исключение любого типа). У метода `start()` есть что-то вроде злого метода-близнеца, называющегося `stop()`, который навсегда убивает поток. Однако этот метод является устаревшим, его не следует использовать. Далее в этой главе мы объясним почему и предложим некоторые более удачные способы остановить поток. Мы также посмотрим на некоторые другие методы, которые вы можете использовать, чтобы управлять продвижением потока, пока он выполняется.

Давайте взглянем на пример. Следующий класс, `Animation`, реализует метод `run()`, чтобы стимулировать его рисовальный список:

```
class Animation implements Runnable {
boolean animate = true;
public void run() {
while ( animate ) {
// нарисовать фреймы
...
}
}
}
```

Чтобы использовать это, мы создаем объект класса `Thread`, передавая ему экземпляр класса `Animation` в качестве его целевого объекта и вызываем его метод `start()`. Мы можем выполнить эти шаги явно:

```
Animation happy = new Animation("Mr. Happy");
Thread myThread = new Thread( happy );
myThread.start();
```

Мы создали экземпляр нашего класса `Animation` и передали его как аргумент конструктору `myThread`. Когда мы вызваем метод `start()`, `myThread` начинает выполнять метод `run()` класса `Animation`. Шоу начинается!

Эта ситуация не является очень объектно-ориентированной. Чаще мы хотим, чтобы объект управлял собственными потоками, как показано на рис. 9.1, изображающем объект `Runnable`, который создает и запускает свой поток. Мы покажем, как наш класс `Animation` выполняет эти действия в конструкторе, хотя на практике, может быть, лучше поместить их в более явный контроллер метода (например, `startAnimation()`):

```
n
class Animation implements Runnable {
Thread myThread;
Animation (String name) {
myThread = new Thread( this );
myThread.start();
}
...
}
```

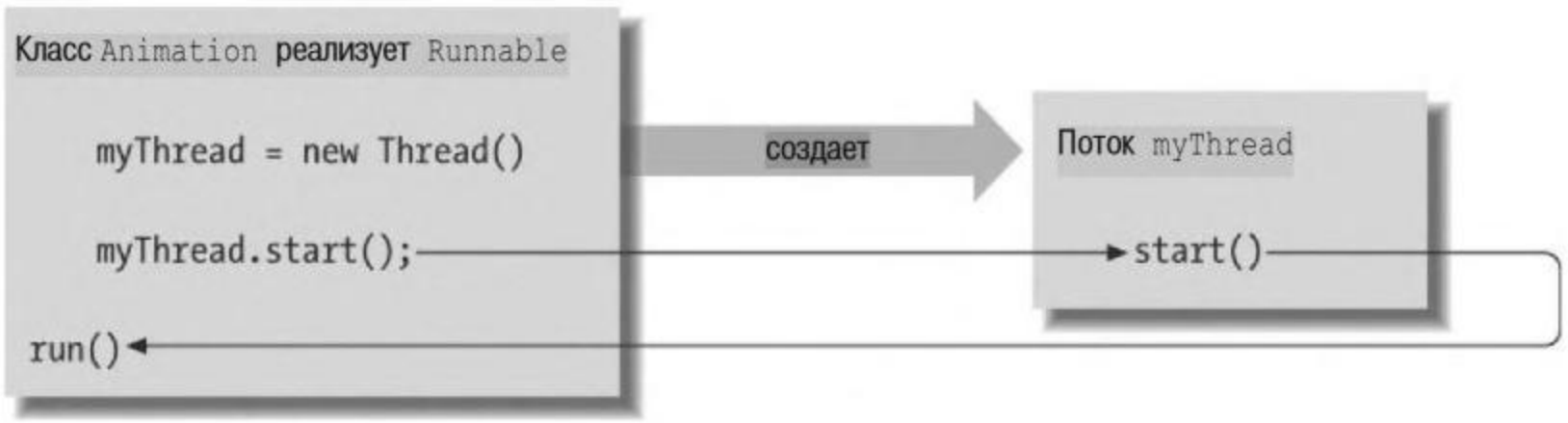

Рис. 9.1. Взаимодействие между классом `Animation` и его потоком

В этом случае аргументом, который мы передаем конструктору `Thread`, является `this`, текущий объект (который является объектом интерфейса `Runnable`). Мы храним ссылку на `Thread` в переменной экземпляра `myThread` в случае, если мы хотим прервать показ или испытать какой-то другой тип управления позже.

Естественно рожденный поток

Интерфейс `Runnable` позволяет нам делать произвольный объект целью потока, как в предыдущем примере. Это наиболее важное общее

использование класса Thread. В большинстве ситуаций, в которых вам нужно использовать потоки, вы создаете класс (возможно, простой класс-адаптер), реализующий интерфейс Runnable.

Однако было бы упущением с нашей стороны не показать вам другую технику создания потока. Другая опция разработки — это сделать наш целевой класс подклассом типа, который уже является выполняемым. Как выясняется, сам класс Thread удобно реализует интерфейс Runnable; он обладает собственным методом run(), который мы можем переопределить непосредственно, чтобы сделать объявление:

```
class Animation extends Thread {
boolean animate = true;
public void run() {
while ( animate ) {
// нарисовать фреймы
...
}
}
}
```

Фреймворк нашего класса Animation выглядит так же, как раньше, кроме того, что наш класс теперь является подклассом Thread. Чтобы следовать этой схеме, дефолтный конструктор класса Thread делает себя дефолтной целью, то есть по умолчанию Thread выполняет собственный метод run(), когда мы вызываем метод start(), как показано на рис. 9.2. Теперь наш подкласс может просто переопределить метод run() в классе Thread. (Сам поток определяет пустой метод run().)

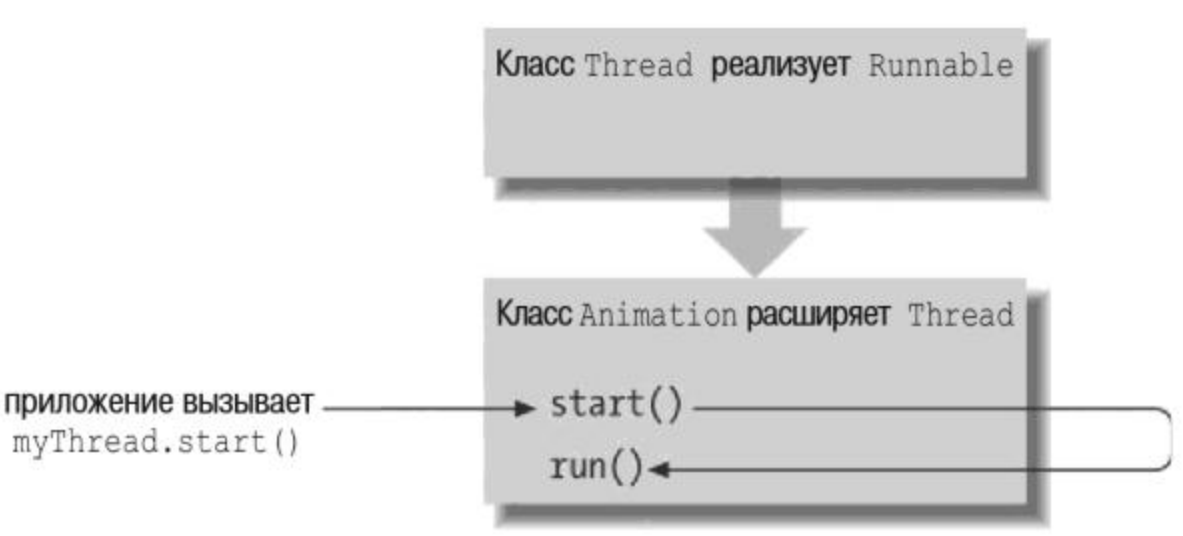

Рис. 9.2. Animation как подкласс Thread

Далее мы создаем экземпляр класса Animation и вызываем его метод start() (который он также унаследовал от класса Thread):

```
Animation bouncy = new Animation("Энергичная");
bouncy.start();
```

Как вариант, мы можем сделать так, чтобы объект `Animation` запустил свой поток при своем создании как ранее:

```
class Animation extends Thread {
Animation (String name) {
start();
}
...
}
```

Здесь наш объект `Animation` вызывает собственный метод `start()`, когда создается экземпляр. (Вероятно, лучше начинать и останавливать наши объекты явно после их создания, а не начинать выполнение потоков как скрытый побочный эффект создания объекта, но для примера это подойдет.)

Создание подклассов класса `Thread` может показаться удобным способом объединения потока и его целевого метода `run()`. Однако этот подход часто не является лучшей моделью. Если вы создаете подкласс класса `Thread`, чтобы реализовать поток, вы заявляете, что вам нужен новый тип объекта, который является подобием класса `Thread`, раскрывающим все публичные API класса `Thread`. Да, есть что-то приятное в том, чтобы брать объект, который изначально связан с выполнением задачи, и делать его объектом класса `Thread`, но фактические ситуации, где вы захотите создать подкласс `Thread`, не слишком распространены. В большинстве случаев более естественно позволить требованиям вашей программы диктовать структуру класса и использовать интерфейсы `Runnable`, чтобы соединить выполнение и логику вашей программы.

Использование адаптера

В конце концов, как мы предполагали, мы можем создать класс-адаптер, который даст нам больше контроля над структурированием кода. Особенно удобно создавать анонимный внутренний класс, который реализует интерфейс `Runnable` и вызывает произвольный метод в нашем объекте. Это очень похоже на запуск потока и указывает произвольному методу выполняться так, как будто у нас есть указатели метода. Например, предположим, что наш класс `Animation` предоставляет метод, называемый `startAnimating()`, который выполняет запуск (за-

гружает изображения и т. д.), а затем запускает поток, чтобы выполнить анимацию. Мы скажем, что фактическое содержимое цикла анимации содержится в приватном методе, называемом `drawFrames()`. Мы можем использовать адаптер для запуска `drawFrames()`.

```
class Animation {
public void startAnimating() {
// выполнять установку, загружать изображения и т. д.
...
// запустить рисующий поток
Thread myThread = new Thread ( new Runnable() {
public void run() { drawFrames(); }
} );
myThread.start();
}
private void drawFrames() {
// сделать анимацию ...
}
```

В этом коде анонимный класс, реализующий интерфейс `Runnable`, сгенерирован для нас компилятором. Мы создаем поток с анонимным объектом в качестве его цели и заставляем его метод `run()` вызывать метод `drawFrames()`. Мы избегаем реализации обобщенного метода `run()` в коде приложения за счет создания дополнительного класса.

Заметьте, что мы могли быть даже более краткими в предыдущем примере, просто сделав так, чтобы наш анонимный внутренний класс расширил, а не реализовал интерфейс `Runnable`. Мы также можем начать новый поток, не сохраняя ссылки на него, если не будем использовать его позже.

Управление потоками

Мы увидели, как метод `start()` использовался для начала выполнения нового потока. Несколько других методов экземпляра позволяют нам явно управлять выполнением потока:

- статический метод `Thread.sleep()` вызывает выполняющийся в текущий момент поток, чтобы ждать определенное количество времени, не потребляя много (или не потребляя вообще) процессорного времени;

- методы `wait()` и `join()` координируют выполнение двух или более потоков. Мы обсудим их подробно, когда будем говорить о синхронизации потоков позже в этой главе;
- метод `interrupt()` включает поток, который спит в цикле `sleep()` или `wait()` или в противном случае блокируется длинной операцией ввода/вывода*.

Устаревшие методы

Мы также должны упомянуть три устаревших метода управления потоками: `stop()`, `suspend()` и `resume()`. Метод `stop()` дополняет `start()`; он разрушает поток. `start()` и устаревший метод `stop()` могут вызываться только один раз за время существования потока. Для сравнения, устаревшие методы `suspend()` и `resume()` использовались для произвольной остановки, а затем перезапуска выполнения потока.

Хотя эти устаревшие методы до сих пор существуют в самой последней версии Java (и, скорее всего, останутся навсегда), их не следует использовать в разработке нового кода.

Проблема методов `stop()` и `suspend()` заключается в том, что они захватывают управление выполнением потока некоординированным, грубым способом. Это делает программирование сложным; приложению не всегда легко предупреждать прерывания в произвольном месте его выполнения и должным образом восстанавливаться после них. Более того, когда поток захватывается с использованием одного из этих методов, система выполнения Java должна освободить все свои внутренние блокировки, используемые для синхронизации потока. Это может вызвать неожиданное поведение и, в случае `suspend()`, легко завести вас в тупик.

Лучший способ влиять на выполнение потока — и требующий только немного больше работы с вашей стороны — это создание некой простой логики в вашем коде потока для использования отслеживающих переменных (флажков), возможно, в объединении с методом `interrupt()`, который позволяет вам пробудить спящий поток. Другими словами, вы должны заставить поток остановить или возобновить то, что он делает, попросив его любезно, а не нарушив его планы неожиданно. В примерах потоков, приведенных в этой книге, тем или иным способом используется эта техника.

---

* `interrupt()` не работал слаженно ни в одной из версий Java.

Метод `sleep()`

Нам часто нужно приказать потоку бездействовать или «спать» в течение определенного периода времени. Пока поток спит или блокируется каким-то другим способом от вводимой информации любого рода, он не потребляет процессорное время и не конкурирует с другими потоками за обработку. Чтобы добиться этого, мы можем вызвать метод `Thread.sleep()`, который воздействует на поток, выполняемый в данное время. Вызов заставляет поток остановить выполнение на определенное количество миллисекунд:

```
try {
// Текущий поток
Thread.sleep( 1000 );
} catch ( InterruptedException e ) {
// кто-то пробудил нас преждевременно
}
```

Метод `sleep()` может выбросить исключение `InterruptedException`, если его прерывает другой поток при помощи метода `interrupt()`. Как видно из примера, поток может отловить это исключение и воспользоваться возможностью выполнить какое-то действие — например, проверить, должна ли существовать переменная, или просто выполнить какую-то организационную операцию, а затем снова отправиться спать.

Метод `join()`

Наконец, если вам нужно координировать вашу деятельность с другим потоком, ожидая, пока он завершит свое задание, вы можете использовать метод `join()`. Вызов метода потока `join()` приводит к блокировке оператора вызова до выполнения потока. В противном случае вы можете запросить поток, вызвав `join()`, со временем ожидания, равным некоторому количеству миллисекунд. Это очень простая форма синхронизации потока. Позже в этой главе мы рассмотрим намного более общий и мощный механизм для координации деятельности потока: `wait()`, `notify()` и даже высокоуровневые API в пакете `java.util.concurrent`.

Метод `interrupt()`

Раньше мы описывали метод `interrupt()` как способ пробудить поток, который бездействует в операторе `sleep()`, `wait()` или длительном операторе ввода/вывода. Любой поток, который не течет постоянно

(не является «жестким циклом»), должен периодически переходить в одно из этих состояний, и, таким образом, это будет пунктом, где поток может быть ослаблен до остановки. Когда поток прерван, устанавливается пометка со *статусом прерывания*. Это может случиться в любое время, вне зависимости от того, бездействует поток или нет. Поток может проверять свой статус при помощи метода `isInterrupted()`. `isInterrupted(boolean)`, другая форма принимает булево значение, указывающее на то, стóит очищать статус прерывания или нет. В этом случае поток может использовать статус прерывания как флаг и сигнал.

Это в действительности предписанная функциональность метода. Однако это всегда было слабым местом, и реализации Java сталкивались с трудностями, заставляя его работать корректно во всех случаях. В более поздних версиях до сих пор есть проблемы с прерыванием вызовов ввода/вывода. Под вызовом ввода/вывода мы подразумеваем ситуацию, когда приложение блокируется методом `read()` или `write()`, перемещая байты к источнику (такому, как файл или сеть) или от него. В этом случае предполагается, что Java выбросит исключение `InterruptedIOException`, когда выполняется метод `interrupt()`. Однако это никогда не было надежным во всех реализациях Java. Для решения проблемы в Java 1.4 был представлен новый фреймворк ввода/вывода (`java.nio`). Когда поток, ассоциируемый с операцией NIO, просыпается, поток ввода/вывода (называемый «каналом») автоматически закрывается. (В главе 12 вы найдете более полную информацию о пакете NIO.)

Смерть потока

Поток продолжает выполняться, пока не происходит что-то из следующего:

- он сам возвращается из метода `run()`;
- он сталкивается с непроверяемым искючением во время выполнения;
- вызывается злой и неприятный устаревший метод `stop()`.

Что происходит, если не происходит ничего из этого и метод `run()` для потока никогда не завершается? На самом деле поток может продолжать существовать даже после того, как то, что якобы является частью создавшего его приложения, завершилось. Это означает, что мы должны знать о том, как как наши потоки в конечном счете завершают-

ся, или приложение может прекратиться, оставив осиротевшие потоки, которые чрезмерно потребляют ресурсы или поддерживают работу приложения, которое в другом случае уже завершилось бы.

Во многих случаях мы действительно хотим создать фоновые потоки, которые выполняют простые периодические задания в приложении. Метод `setDaemon()` можно использовать, чтобы пометить поток как потоковый демон, который должен быть отброшен, когда не остается никаких других не демонических потоков. Обычно интерпретатор Java продолжает работать, пока не выполнятся все потоки. Но когда останутся только потоковые демоны, интерпретатор выйдет.

Вот дьявольский пример, в котором используются потоковые демоны:

```
class Devil extends Thread {
Devil() {
setDaemon( true );
start();
}
public void run() {
// выполняет злые задания
}
}
```

В этом примере поток `Devil` устанавливает статус «демон», когда он создается. Если какие-то потоки `Devil` остаются, когда в противном случае приложение было бы закрыто, система выполнения убивает их за нас. Мы не должны беспокоиться об их очистке.

Потоки-демоны изначально полезны в автономных приложениях Java и реализации серверных фреймворков, но не в таких компонентных приложениях, как апплеты. Поскольку апплет выполняется внутри другого приложения Java, любые потоковые демоны, которых он создает, могут продолжать существовать, пока контролирующее приложение не выйдет — вероятно, не то, чего вы хотите. Браузер или любое другое приложение может использовать `ThreadGroups`, чтобы вместить все потоки, созданные подсистемой приложения, и может очистить их при необходимости.

И напоследок расскажем об изящном уничтожении потоков. Очень распространенной проблемой, с которой сталкиваются новые разработ-

чики, когда первый раз создают приложение, используя компонент AWT или Swing, является то, что их приложение никогда не выходит; виртуальная машина зависает, когда все завершается. При работе с графикой Java создает поток AWT для обработки потока и событий рисования. Поток AWT не является потоковым демоном, поэтому он автоматически не выходит, когда другие приложения потока завершились, и разработчик должен явно вызывать `System.exit()`. (Если подумать, это имеет смысл. Поскольку многие приложения с графическим интерфейсом пользователя управляются событиями и просто ожидают ввода данных пользователем, они бы просто вышли после того, как их код установки завершится.)

Выполнение апплета в нескольких потоках

Апплеты являются приложениями Java встроенной конструкции, которые могут запускаться и останавливаться по команде много раз за время их существования. Веб-браузер с установленным Java обычно запускает апплет, когда апплет показывается, и завершает его, когда пользователь переходит на другую страницу или (теоретически), когда пользователь прокручивает страницу, убирая апплет с экрана. Согласно этому интерфейсу API, мы хотели бы, чтобы апплет прекратил свою несущественную деятельность, когда он останавливается и продолжал ее при новом запуске. Мы поговорим об апплетах в главе 23, но в действительности не так важно знать о них сейчас. Мы просто используем это как более реалистичный пример и как переход к разговору о нашей следующей теме, синхронизации.

В этой главе мы создадим `UpdateApplet`, простой базовый класс для приложения, который управляет потоком, чтобы автоматически обновлять его показ через регулярные промежутки. `UpdateApplet` управляет базовым созданием и прекращением потока в методах апплета `start()` и `stop()`:

```
public class UpdateApplet extends java.applet.Applet
implements Runnable
{
Thread thread;
boolean running;
int updateInterval = 1000;
public void run() {
```

```
while ( running )
{
repaint();
try {
Thread.sleep( updateInterval );
} catch ( InterruptedException e ) {
System.out.println("interrupted...");
return;
}
}
}
public void start() {
System.out.println("starting...");
if ( !running ) // наивный подход
{
running = true;
thread = new Thread(this);
thread.start();
}
}
public void stop() {
System.out.println("stopping...");
thread.interrupt();
running = false;
}
}
```

`UpdateApplet` является объектом интерфейса `Runnable`, который попеременно спит и вызывает метод `repaint()`. (Однако рисовать нечего, поэтому запуск этого апплета является довольно скучным. Позже мы создадим от него подкласс, чтобы реализовать цифровые часы.) Он имеет два других публичных метода: `start()` и `stop()`. Это методы класса `Applet`, которые мы переопределяем; не путайте их с методами класса `Thread` с такими же именами. Эти методы `start()` и `stop()` вызываются веб-браузером или обозревателем апплетов, чтобы сообщить апплету, когда он должен и не должен выполняться.

`UpdateApplet` иллюстрирует дружественный среде способ работы с потоками в простом апплете. `UpdateApplet` просто прекращает свой поток каждый раз, когда апплет останавливается, и воссоздает, когда апплет перезапускается. Когда вызывается метод `start()` класса `UpdateApplet`,

мы сначала убеждаемся, что нет текущего выполняющегося потока, проверяя метку `running`. Мы создаем ее, чтобы начать выполнение. Когда наш апплет впоследствии получает команду об остановке, мы устанавливаем метку, указывающую, что он должен остановиться, и убеждаемся, что поток действует, вызывая метод `interrupt()`. Таким способом мы точно отлавливаем поток в начале его следующей итерации или когда он засыпает.

Имея класс `UpdateApplet`, который делает всю работу за нас, мы можем создать апплет самых простых часов в мире всего несколькими строчками кода. Рис. 9.3 демонстрирует апплет `Clock`.



Рис. 9.3. Апплет `Clock`

Ниже представлен код:

```
//файл: Clock.java
public class Clock extends UpdateApplet {
public void paint( java.awt.Graphics g ) {
g.drawString( new java.util.Date().toString(), 10, 25 );
}
}
```

Метод `java.util.Date().toString()` создает строку, которая содержит текущее время.

Отслеживание проблем

Наш апплет кажется довольно простым и в действительности работает, как было разрекламировано. Но кое-что в нем должно нас обеспокоить, когда мы думаем о потоках. Давайте посмотрим на эту быструю проверку флага `running` прежде, чем начнем наш новый поток:

```
if ( !running ) // родной подход
{
running = true;
... /* начать поток */
```

Теперь методы `start()` и `stop()` класса `Applet` гарантированно вызываются последовательно и, вероятно, тем же конролирующим потоком. В результате эта проверка на существование рабочего потока в `start()` может показаться ненужной здесь. Метод `stop()` всегда должен вызваться перед тем, как снова будет вызван `start()`. Но в стиле защитного программирования тест кажется хорошей вещью, правда? Может быть и так, но в основном этого не достаточно, чтобы предотвратить плохие события. Тест может предотвратить простой случай смещенных вызовов `stop()` и `start()`, но здесь скрывается большой вопрос: что происходит, если `start()` и `stop()` вызываются повторно или в очень быстрой последовательности в многопоточной среде? В крайнем случае будет возможным для двух потоков войти в тест почти в одно время, и есть вероятность, что в результате будет начато много неконтролируемых потоков. Нужен реальный способ получить эксклюзивный доступ к ресурсу (или метке) на какое-то время. Это то, для чего нужна синхронизация, и мы рассмотрим ее в деталях в следующей главе и в оставшейся части этой главы.

С синхронизацией мы также должны учитывать более сложные сценарии нашего апплета, такие как поддержание работы нашего потока и отправка его в спячку, когда апплет остановлен. Это позволило бы нам предотвратить такую затратную установку, как сетевые соединения, и избавиться от них позже при необходимости.

Синхронизация

У каждого потока есть свой разум. Обычно поток занимается собственными делами, не учитывая то, чем занимаются другие потоки в приложении. Потоки могут иметь интервалы во времени, это означает, что они могут работать произвольными импульсами и толчками, как будет указано операционной системой. В многопроцессорной системе даже возможно одновременное выполнение разных потоков в разных центральных процессорах. Эта глава посвящена координации деятельности двух или более потоков таким образом, чтобы они могли работать совместно и не конфликтовать из-за использования одинаковых переменных и методов (координируя их игру на поле для гольфа).

Язык Java предоставляет несколько простых структур для синхронизации деятельности потоков. Они все основаны на идее контролирующих программ, широко используемых в схеме синхронизации. Вам не нужно знать подробности того, как работают контролирующие програм-

мы, чтобы уметь использовать их, но это поможет вам иметь в голове полную картину.

Контролирующая программа — это изначально блокировка. Блокировка прикреплена к ресурсу, которому может быть необходим доступ многих потоков, но к нему должен обращаться только один поток в единицу времени. Это очень похоже на комнату отдыха с замком на двери; если она не заперта, вы можете войти и запереть ее, пока будете в комнате. Если ресурс не используется, поток может получить блокировку и обратиться к ресурсу. Когда поток выполнен, он освобождает блокировку, как вы открываете дверь комнаты отдыха и оставляете ее открытой для следующего человека. Однако, если у другого потока уже есть блокировка для ресурса, все другие потоки должны ждать, пока текущий поток не выполнится и не освободит блокировку. Как будто комната для отдыха занята, когда вы приходите: вам нужно ждать, пока текущий пользователь закончит свои дела и откроет дверь.

К счастью, Java делает процесс синхронизации доступа к ресурсам достаточно простым. Язык управляет установкой и приобретением замков; все, что вам нужно, это указать ресурсы, которые нуждаются в синхронизации.

Синхронизированный доступ к методам

Наиболее распространенная потребность в синхронизации среди потоков в Java — это упорядочивание их доступа к определенному ресурсу (объекту) — другими словами, обеспечение того, что только один поток в момент времени может управлять объектом или переменной*. В языке Java у каждого объекта есть сопутствующая блокировка. Чтобы быть более точными, у каждого класса и экземпляра класса есть своя блокировка. Ключевое слово `synchronized` помечает места, где поток должен получить блокировку перед тем, как продолжить.

Например, предположим, что мы реализовали класс `Speech Synthesizer`, который содержит метод `say()`. Мы не хотим, чтобы множество потоков вызывало `say()` в одно и то же время, потому что мы не сможем понять ничего из того, что будет сказано. Поэтому мы помечаем метод

* Не путайте понятие «сериализовать» в этом контексте с сериализацией объектов Java, которая является механизмом для того, чтобы делать объекты постоянными. Однако изначальный смысл (поместить одну вещь после другой) относится к обоим. В случае с сериализацией объекта данные объекта устанавливаются байт за байтом в определенном порядке.

say() как synchronized, это означает, что поток должен получить блокировку к объекту SpeechSynthesizer перед тем, как он может заговорить:

```
class SpeechSynthesizer {
synchronized void say( String words ) {
// говорить
}
}
```

Поскольку say() является методом экземпляра, поток должен получить блокировку к экземпляру SpeechSynthesizer, который он использует перед тем, как вызвать метод say(). Когда say() выполнился, он оставляет блокировку и выполняет метод. Неважно, принадлежит поток самому SpeechSynthesizer или какому-то другому объекту; каждый объект должен получить одну и ту же блокировку к экземпляру SpeechSynthesizer. Если бы say() был методом класса (статическим методом), а не методом экземпляра, мы могли бы пометить его как synchronized. В этом случае, поскольку не вовлечен объект экземпляра, блокировка находится на самом объекте.

Часто вы хотите синхронизировать множество методов того же класса так, чтобы только один метод изменял или исследовал части класса в данный период времени. Все статические синхронизованные методы в классе используют ту же блокировку объекта экземпляра. С одним и тем же токеном все методы экземпляра в классе используют ту же блокировку объекта экземпляра. Таким образом, Java может гарантировать, что только один набор синхронизированных методов работает в определенное время. Например, класс SpreadSheet может содержать набор переменных экземпляра, которые представляют значения ячеек, а также некоторые методы, которые управляют ячейками в ряду:

```
class SpreadSheet {
int cellA1, cellA2, cellA3;
synchronized int sumRow() {
return cellA1 + cellA2 + cellA3;
}
synchronized void setRow( int a1, int a2, int a3 ) {
cellA1 = a1;
cellA2 = a2;
cellA3 = a3;
}
...
}
```

В этом примере методы `setRow()` и `sumRow()` оба обращаются к переменным ячеек. Как видите, проблемы могут возникнуть, если один поток изменял значения переменных в `setRow()` в тот самый момент, когда другой поток считывал значения в `sumRow()`. Чтобы предотвратить это, мы пометили оба метода как `synchronized`. Когда потоки синхронизированы, только один выполняется в определенное время. Если поток находится в середине выполнения `setRow()`, когда другой поток вызывает `sumRow()`, второй поток ожидает, пока первый закончит выполнение `setRow()` перед тем, как будет выполнять `sumRow()`. Эта синхронизация позволяет нам сохранить последовательность класса `SpreadSheet`. По большей части все это блокирование и ожидание управляется Java и остается невидимым для программиста.

В дополнение к синхронизации всех методов ключевое слово `synchronized` может использоваться в специальной конструкции, чтобы предохранять произвольные блоки кода. В этой форме ключевое слово также принимает явный аргумент, указывающий объект, для которого нужно получить блокировку:

```
synchronized( myObject) {
//Функционал, который нуждается в эксклюзивном доступе к ресурсам
}
```

Этот блок кода может появляться в любом методе. Когда он достигается, поток должен получить блокировку к `myObject` перед тем, как продолжить. Таким образом, мы можем синхронизировать методы (или части методов) в разных классах таким же образом, как методы в одном классе.

Следовательно, синхронизированный метод экземпляра эквивалентен методу с операторами, синхронизированными для текущего объекта. Соответственно:

```
synchronized void myMethod () {
...
}
```

Эквивалентно:

```
void myMethod () {
synchronized ( this ) {
...
}
}
```

Класс доступа и переменные экземпляров объекта из потоков многопоточных программ

В примере с классом `SpreadSheet` мы защищали доступ к набору переменных с синхронизированным методом, чтобы избежать смены одной из переменных, пока считываются другие. Мы хотели их координировать. Но что насчет индивидуальных типов переменных? Нуждаются ли они в синхронизации? Обычно ответ «нет». Почти все операции над простыми типами и объектами ссылочного типа в Java происходят *автоматически*, то есть они регулируются виртуальной машиной в один шаг, без единого шанса того, что два потока столкнутся. Это предотвращает обращение потоков к ссылкам, пока они находятся в процессе обращения к ним других потоков.

Но остерегайтесь — мы сказали *почти*. Если вы внимательно прочтете спецификацию виртуальной машины, вы увидите, что автоматическая обработка примитивных типов `long` не гарантирована. Оба этих типа представляют 64-битные значения. Проблема заключается в том, как стек виртуальной машины обрабатывает их. Возможно, что эта спецификация будет совершенствоваться. Но сейчас вы должны синхронизировать доступ к вашим экземплярам переменных типов `double` и `long` через методы доступа или использовать ключевое слово `volatile` или атомарный класс-обертку, который мы опишем позже.

Другая проблема, не зависящая от атомарности значений, это концепция различных потоков в виртуальной машине, кеширующей значения на определенный период времени — то есть, даже если один поток мог изменить значение, виртуальная машина может не быть обязанной делать так, чтобы это значение появилось, пока виртуальная машина не достигнет определенного состояния, известного как «барьер памяти». Поскольку это не должно быть проблемой в большинстве случаев реального программирования, вы можете справиться с этим, объявив переменную с ключевым словом `volatile`. Ключевое слово указывает виртуальной машине, что значение может быть изменено внешними потоками, и эффективно синхронизирует доступ к нему автоматически.

В конце концов, `java.util.concurrent.atomic` предоставляет синхронизированные классы-оболочки всем простым типам и ссылкам. Эти оболочки предоставляют не только простые операции над значениями `set()` и `get()`, но и специализированные «комбинированные» операции, такие как `compareAndSet()`, которая работает автоматически и может использоваться для создания компонентов синхронизированного прило-

жения высшего уровня. Классы в этом пакете были разработаны специфически, чтобы составить схему функционала на уровне программного обеспечения во многих случаях и могут быть очень эффективными. Мы поговорим о них подробнее позже в этой главе.

Повторное получение блокировки

Блокировки, полученные Java при входе в синхронизированный метод или блок кода, являются *входящими*. Это означает, что поток, держащийся за блокировку, может получать такую же блокировку снова любое количество раз и никогда не блокируется, ожидая себя. В большинстве случаев это означает, что код ведет себя так, как вы ожидаете; поток может вызывать синхронизированный метод рекурсивно и сам может вызывать другие синхронизированные методы внутри того же объекта.

Методы `wait()` и `notify()`

С ключевым словом `synchronized` мы можем упорядочивать выполнение методов и блоков кода, так что только один поток в единицу времени может выполнять синхронизированный элемент. Методы `wait()` и `notify()` класса `Object` расширяют эту способность, позволяя нам явно координировать ожидающие и выполняющиеся потоки. Каждый объект в Java является подклассом класса `Object`, поэтому каждый объект наследует эти методы. Используя `wait()` и `notify()`, поток может эффективно завершить свое удержание блокировки в произвольный момент, а затем ожидать другой поток, чтобы вернуть ее перед продолжением. Вся координированная деятельность все еще происходит в синхронизированных блоках и по-прежнему только один поток выполняется в определенный момент времени.

Выполняя метод `wait()` из синхронизированного блока, поток освобождает свое удержание блокировки и отправляется спать. Поток может делать это, если ему нужно ждать чего-то, что произойдет в другой части приложения, как мы увидим вскоре. Позже, когда необходимое событие происходит, выполняющийся поток вызывает `notify()` из блока, синхронизированного на том же объекте. Первый поток просыпается и начинает пытаться снова приобрести блокировку. Когда первому потоку это удается, он продолжает оттуда, где он был оставлен. Однако поток, который находился в ожидании, может не получить блокировку немедленно (а может и вовсе ее не получить). Это зависит от того, где второй поток в конце концов отпустит блокировку и какому потоку

удастся зацепиться за нее снова. Первый метод не пробудится от `wait()`, пока другой поток не вызовет `notify()`. Однако перегруженная версия `wait()` позволяет нам определить период ожидания. Если другой поток не вызовет `notify()` в указанный срок, ожидающий поток автоматически просыпается.

Давайте посмотрим на простой сценарий, чтобы увидеть, что происходит. В следующем примере мы предполагаем, что существует три потока — один ожидает выполнения каждого из трех синхронизированных методов класса `MyThing`. Мы назовем их *ожидающими*, *уведомляющими* и *взаимосвязанными* потоками. Вот фрагмент кода для иллюстрации:

```
class MyThing {
synchronized void waiterMethod() {
// do some stuff сделать какую-то вещь
wait(); // теперь ожидать, что уведомляющий поток что-то сделает
// продолжить там, где мы закончили
}
synchronized void notifierMethod() {
// do some stuff сделать какую-то вещь
notify(); // уведомить ожидающий поток, что мы это сделали
// продолжать что-то делать
}
synchronized void relatedMethod() {
// сделать какую-то родственную вещь
}
...
}
```

Давайте предположим, что поток, названный *ожидающим*, проходит через ворота первым и начинает выполнять метод `waiterMethod()`. Два других потока изначально блокируются при попытке получить блокировку объекта `MyThing`. Когда *ожидающий* метод выполняет метод `wait()`, он освобождает свое удержание блокировки и отправляется спать. Теперь два конкурентных потока ожидают блокировки. Какой поток получает ее, зависит от нескольких факторов, включая вероятность и приоритеты потоков. (Мы обсудим планирование потоков в следующей главе.)

Давайте предположим, что *уведомляющий* поток является следующим потоком, который приобретает блокировку, поэтому он начинает выполнять метод `notifierMethod()`. *Ожидающий* поток продолжает

спать, а *связанный* теряет силы, ожидая своей очереди. Когда *уведомляющий* метод выполняет вызов метода `notify()`, система выполнения подталкивает *ожидающий* поток, по сути, сообщая ему, что что-то изменилось. *Ожидающий* поток просыпается и возвращается к сообщающему, соперничая за блокировку `MyThing`. Он не получает блокировку автоматически; он просто меняет свое состояние с «Оставьте меня в покое» на «Я хочу блокировку».

В этот момент *нотифицирующий* метод все еще владеет блокировкой и продолжает удерживать ее, пока не возвращается синхронизированный метод `notifierMethod()` или, возможно, сам выполняет метод `wait()`. В этот момент два других потока борются за блокировку. *Ожидающий* поток хочет продолжать метод ожидающего `Method()` с того места, где он закончил, в то время как *связанный*, который терпеливо ожидал, хочет начать. Мы позволим вам выбрать собственное окончание истории.

Для каждого вызова метода `notify()` система выполнения пробуждает только один поток, который спит в вызове метода `wait()`. Группа потоков, ожидающих блокировки, называется *ожидающим набором*. Если ожидает множество потоков, Java выбирает поток произвольно, что может зависеть от реализации. Класс `Object` также предоставляет вызов метода `notifyAll()`, чтобы пробудить все спящие потоки. В большинстве случаев вы, вероятно, захотите использовать `notifyAll()`, а не `notify()`. Запомните, что `notify()` в действительности означает: «Эй, что-то связанное с этим объектом изменилось. Условие, которого вы ожидаете, возможно, изменилось, поэтому проверьте его снова». В общем, нет причин предполагать, что только один поток в момент времени заинтересован в изменении или способен воздействовать на него. Различные потоки могут по-разному смотреть на то, что могло измениться.

Условия ожидания

В общем, наш *ожидающий* поток ожидает отдельного условия для изменения, и мы захотим посадить его в цикл таким образом:

```
while ( condition != true )
wait();
```

Эта проверка называется *условием ожидания*. Другие синхронизированные потоки вызывают метод `notify()` или метод `notifyAll()`, когда они изменили среду, так что условие может быть проверено снова. Важ-

но использовать цикл на условии ожидания, чтобы убедиться, что поток пробужден по правильной причине. Потоки также могут использовать рассчитанную во времени версию метода `wait()`, чтобы совершить периодическую работу, проверяя условие таким образом. Использование таких условий ожидания является также альтернативой вызова и сна, как вы увидите в следующей главе.

Передача сообщений

Далее мы проиллюстрируем классическое взаимодействие между двумя потоками: `Producer` и `Consumer`. Поток-источник создает сообщения и помещает их в очередь, пока потребитель читает и показывает их. Чтобы быть реалистами, мы сделаем наш поток-потребитель ленивым, и он будет выполняться намного медленнее, чем источник. Это означает, что поток `Producer` время от времени должен останавливаться и ждать, чтобы `Consumer` догнал его. Пакет согласованности Java имеет интерфейс `BlockingQueue`, который предоставляет именно этот тип функционала, но мы создадим его сами здесь, используя сначала базовые технологии синхронизации, а затем посмотрим на классы `Queue` и все библиотеки классов в главе 11.

Ниже представлены классы `Producer` и `Consumer`:

```
import java.util.*;
public class Consumer implements Runnable {
Producer producer;
Consumer( Producer producer ) {
this.producer = producer;
}
public void run() {
while ( true ) {
String message = producer.getMessage();
System.out.println("Got message: " + message);
try {
Thread.sleep( 2000 );
} catch ( InterruptedException e ) { }
}
}
public static void main(String args[]) {
Producer producer = new Producer();
new Thread( producer ).start();
```

```
Consumer consumer = new Consumer( producer );
new Thread( consumer ).start();
}
}
public class Producer implements Runnable{
static final int MAXQUEUE = 5;
private List messages = new ArrayList();
public void run() {
while ( true ) {
putMessage();
try {
Thread.sleep( 1000 );
} catch ( InterruptedException e ) { }
}
}
// called by Producer internally
private synchronized void putMessage()
{
while ( messages.size() >= MAXQUEUE )
try {
wait();
} catch( InterruptedException e ) { }
messages.add( new java.util.Date().toString() );
notify();
}
// called by Consumer externally
public synchronized String getMessage()
{
while ( messages.size() == 0 )
try {
notify();
wait();
} catch( InterruptedException e ) { }
String message = (String)messages.remove(0);
notify();
return message;
}
}
```

Для удобства мы включили метод `main()` в класс `Consumer`, который выполняет полный пример. Он создает класс `Consumer`, который связан

с классом Producer, и запускает оба класса. Вы можете запустить пример следующим образом:

```
% java Consumer
```

Это создает сообщения с временными метками, созданные классом Producer:

```
Got message: Sun Dec 19 03:35:55 CST 2006
Got message: Sun Dec 19 03:35:56 CST 2006
Got message: Sun Dec 19 03:35:57 CST 2006
...
```

Временные метки изначально показывают интервал в одну секунду, хотя они появляются каждые две секунды. Наш класс Producer выполняется быстрее, чем Consumer. Producer хочет генерировать новое сообщение каждую секунду, в то время как Consumer добирается до чтения и вывода сообщения только каждые две секунды. Видите, сколько времени уходит у очереди сообщения на заполнение? Что происходит, когда она заполнилась?

Давайте посмотрим на код. Здесь мы используем несколько новых инструментов. Producer и Consumer реализуют интерфейс Runnable, и у каждого есть поток, связанный с ним. Классы Producer и Consumer передают сообщения через экземпляр объекта java.util.List. Мы еще не обсуждали класс List, но изначально он является динамическим массивом элементов. Мы используем его как очередь, просто добавляя и убирая элементы в порядке поступления/убывания. List не имеет максимальной емкости, но мы налагаем ее своей проверкой. Важная деятельность происходит в синхронизированных методах: putMessage() и getMessage(). Хотя один из методов используется в потоке Producer, а другой в потоке Consumer, они оба существуют в классе Producer, так что мы можем координировать их, просто объявляя их synchronized. Здесь они оба косвенно используют блокировку объекта Producer. Если очередь пуста, Consumer блокируется в вызове в Producer, ожидая другого сообщения.

Другой вариант разработки реализует метод getMessage() в классе Consumer и использует блок кода synchronized, чтобы явно синхронизировать объект Producer. В любом случае синхронизация на Producer обеспечивает то, что мы имеем множество объектов Consumer, которые питаются от того же класса Producer. Мы сделаем это позже в этой главе.

Работа putMessage() заключается в добавлении нового сообщения в очередь. Он не может сделать это, если очередь уже заполнена, поэто-

му он первым делом проверяет количество элементов в классах message. Если есть место, он вставляет другое сообщение с временной отметкой. Однако если очередь заполнена, putMessage() должен ждать, пока появится место. В этой ситуации putMessage() выполняет wait() и полагается на потребителя, который вызовет notify(), и тот разбудит его после того, как сообщение будет прочтено. Здесь putMessage() проверяет условие в цикле. В этом простом примере проверка может казаться необязательной; мы можем допустить, что, когда putMessage() просыпается, есть свободное место. Однако важно всегда проверять наше условие ожидания в таком цикле, когда мы синхронизируем потоки, поскольку иначе нельзя быть уверенными в том, почему наш поток пробудился. Прежде чем он закончится, putMessage() сам вызывает notify(), чтобы подстегнуть любой Consumer, который может ожидать в пустой очереди.

Метод getMessage() получает сообщение для Consumer. Он входит в цикл метода putMessage(), ожидая, пока в очереди будет хотя бы один элемент прежде, чем продолжить. Если очередь пуста, он выполняет метод wait() и ожидает, что Producer вызовет метод notify(), когда больше элементов будут доступными. Отметьте, что getMessage() делает собственные вызовы notify(). Он делает это каждый раз, когда очередь пуста, чтобы подтолкнуть поток-источник, который может спать и после того, как он поглотит сообщение, чтобы дать источнику разрешение на заполнение очереди снова. Эти сценарии наиболее правдоподобны, если есть больше потребителей, как мы увидим далее.

Давайте добавим еще один потребитель к сценарию, просто чтобы стало поинтереснее. Большинство необходимых изменений находятся в классе Consumer; вот код измененного класса, теперь называемого NamedConsumer:

```
public class NamedConsumer implements Runnable
{
Producer producer;
String name;
NamedConsumer(String name, Producer producer) {
this.producer = producer;
this.name = name;
}
public void run() {
while ( true ) {
String message = producer.getMessage();
System.out.println(name + " got message: " + message);
```

```
try {
Thread.sleep( 2000 );
} catch ( InterruptedException e ) { }
}
}
public static void main(String args[]) {
Producer producer = new Producer();
new Thread( producer ).start();
NamedConsumer consumer = new NamedConsumer( "One", producer );
new Thread( consumer ).start();
consumer = new NamedConsumer( "Two", producer );
new Thread( consumer ).start();
}
}
```

Конструктор `NamedConsumer` использует строку, чтобы идентифицировать каждого потребителя. Метод `run()` использует это имя в вызове `println()`, чтобы идентифицировать, какой потребитель получил сообщение.

Единственное необходимое изменение кода `Producer` заключается в замене вызовов `notify()` на вызовы `notifyAll()` в `putMessage()` и `getMessage()`. (Мы могли бы использовать `notifyAll()` на первом месте.) Теперь вместо источника и потребителя, играющих в пятнашки с очередью, может быть много игроков, ожидающих условия изменения очереди. У нас может быть некоторое количество потребителей, ожидающих сообщения, или источник, ожидающий, когда потребитель примет сообщение. Каждый раз, когда меняется условие очереди, мы заставляем все ожидающие методы переоценить ситуацию, вызвав `notifyAll()`.

Ниже представлен образец вывода, где два `NamedConsumer` выполняются как в методе `main()`, показанном ранее:

```
One got message: Sat Mar 18 20:00:01 CST 2006
Two got message: Sat Mar 18 20:00:02 CST 2006
One got message: Sat Mar 18 20:00:03 CST 2006
Two got message: Sat Mar 18 20:00:04 CST 2006
One got message: Sat Mar 18 20:00:05 CST 2006
Two got message: Sat Mar 18 20:00:06 CST 2006
One got message: Sat Mar 18 20:00:07 CST 2006
Two got message: Sat Mar 18 20:00:08 CST 2006
...
```

Мы видим хорошее, аккуратное чередование между двумя потребителями в результате обращений к `sleep()` в различных методах. Однако произошли бы интересные вещи, если бы мы убрали все обращения к `sleep()` и позволили всему работать на полной скорости. Потоки бы конкурировали и их поведение зависело бы от того, использует ли система временные интервалы. В системе с временными интервалами должно быть достаточно случайное распределение между двумя потребителями, в то время как в системе без временных интервалов единственный потребитель может монополизировать сообщения. Мы вскоре поговорим о том, как потоки соревнуются за время, когда будем обсуждать приоритеты потоков и их планирование.

Пища для размышлений

В этом простом примере многое можно доработать. То, что мы пытались подчеркнуть, — это защитный стиль программирования с учетом нотификаций по потокам. Вам нужно полагаться на условия реального мира, которые вы можете проверять при синхронизации потоков; несложно предположить, что вы получите нужные нотификации в нужном месте в нужное время. Принимая во внимание вышесказанное, наш пример генерирует внешние нотификации, которые устанавливают потоки и тогда, когда для них может не быть работы. Например, мы генерируем нотификации и когда очередь пустая, и когда полная. Лучше было бы разделить эти случаи и использовать две разные блокировки объектов. К счастью, большинству программистов не придется иметь дело с проблемами на этом уровне, тем более что Java предоставляет реальные классы `Queue` и другие конструкции синхронизации высокого уровня.

Объекты `ThreadLocal`

Распространенной проблемой является потребность сохранения некоторой информации или состояния на поточной базе. Например, мы можем захотеть перенести некоторый контекст с текущим потоком, когда он выполняет наше приложение. Или мы можем просто захотеть иметь значение, которое будет различным для различных потоков таким же образом, как каждый поток «видит» собственные локальные переменные в методе. Java поддерживает это через класс `ThreadLocal`. `ThreadLocal` является оберткой объекта, которая автоматически сохраняет отдельное значение для каждого потока, вызывающего его. Например:

```
ThreadLocal userID = new ThreadLocal();
userID.set("Pat"); // вызванный потоком 1
userID.set("Bob"); // вызванный потоком 2
userID.get(); // поток 1 получает "Pat"
userID.get(); // поток 2 получает "Bob"
```

Вы можете использовать экземпляр `ThreadLocal` везде, где вы можете использовать статическую переменную или переменную экземпляра, чтобы автоматически сохранять отдельные значения для каждого потока. Вы также можете расширять `ThreadLocal` и переопределять его метод `initialValue()`. `ThreadLocal` затем будет использовать этот метод для инициализации его значения, когда впервые вызывается метод `get()`:

```
class MyThreadLocalFactory extends ThreadLocal<Factory> {
protected Factory initialValue() { return new MyFactory(); }
}
```

Классы `ThreadLocal` реализуются при использовании класса `Map`, прикрепленного к каждому экземпляру `Thread`, так что их значения исчезают, когда `Thread` больше не используется и собирается мусор.

Полезным дополнением в Java 7 стал класс `ThreadLocalRandom`, который является расширением класса `java.util.Random`, обсуждаемого в главе 11. Класс `ThreadLocalRandom` ликвидирует разногласие (ожидание в связи с синхронизацией) генератора случайных чисел при вызове из разных потоков.

Планирование и приоритеты

Java дает некоторые гарантии относительно того, как он планирует потоки. Почти все планирование потоков Java оставлено реализации Java и в какой-то степени приложению. Хотя, возможно, имело бы смысл (и уж точно сделало бы многих разработчиков счастливее), если бы разработчики Java точно определили алгоритм планирования, но один алгоритм не обязательно подошел бы всем ролям, которые может выполнять Java. Вместо этого разработчики Java накладывают на вас бремя написания грубого кода, который работает, не обращая внимания на алгоритм планирования, и позволяет реализации настраивать алгоритм для лучшей подгонки*.

* Достойной внимания альтернативой этому является спецификация Java в реальном времени, которая определяет специализированное поведение потока для определенных типов приложения. Она разработана в сообществе Java и может быть найдена по адресу **rtsj.dev.java.net**.

Приоритетные правила, которые мы опишем далее, тщательно сформулированы в спецификации языка Java, чтобы быть общим руководством для планирования потока. Вы можете положиться на это поведение в целом (статистически), но писать код, который полагается на очень специфические характеристики планировщика, чтобы работать правильно, не очень хорошая идея. Вместо этого вы должны использовать инструменты управления и синхронизации, которые мы описали в этой главе для управления вашими потоками*.

Каждый поток имеет значение приоритета. В общем, каждый раз, когда поток более высокого приоритета, чем текущий, становится выполняемым (запускается, прекращает спать или уведомляется), он вытесняет потоки более низкого приоритета и начинает выполнение. По умолчанию потоки с одинаковым приоритетом планируются циклически. Это означает, что если поток начал выполнение, он продолжает его до тех пор, пока не начнет делать что-то из следующего:

- спать, вызвав `Thread.sleep()` или `wait()`;
- ждать блокировки, чтобы выполнить метод `synchronized`;
- блокировать информацию ввода/вывода, например при вызове `read()` или `accept()`;
- явно выходить из-под контроля, вызывая `yield()`;
- прекращаться, выполнив свой целевой метод или с вызовом `stop()` (устарело).

Эта ситуация показана на рис. 9.4:

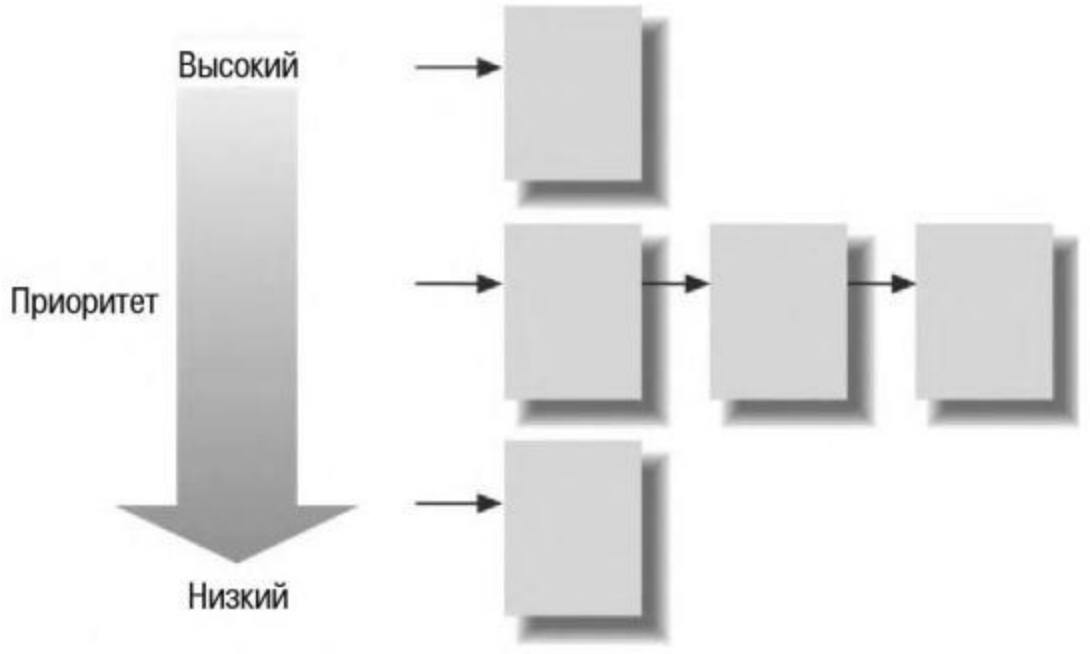

Рис. 9.4. Циклическое планирование с вытесняющим приоритетом

* Книга Java Threads, написанная Скоттом Оуксом и Генри Вонгом (O'Reilly, 2004), включает подробное описание синхронизации, планирования и других вопросов, связанных с потоками.

Состояние потока

В любой момент времени поток находится в одном из пяти общих состояний, которые охватывают его жизненный цикл и действия.

Эти состояния определены в перечислении `Thread.State` и запрошены методом `getState()` класса `Thread`.

`NEW`

Поток был создан, но еще не запущен.

`RUNNABLE`

Нормальное активное состояние выполняющегося потока, включая время, когда поток блокируется в операции ввода/вывода, как чтение, письмо или сетевое соединение.

`BLOCKED`

Поток заблокирован, ожидая входа в синхронизированный метод или блок кода. Это включает время, когда поток был пробужден `notify()` и когда пытается приобрести блокировку после метода `wait()`.

`WAITING, TIMED_WAITING`

Поток ожидает другой поток через вызов `wait()` или `join()`. В случае `TIMED_WAITING` вызов имеет задержку.

`TERMINATED`

Поток выполнен, благодаря возврату, исключению или прекращению.

Мы можем показать состояние всех потоков Java (в текущей группе потоков) следующим отрывком кода:

```
Thread [] threads = new Thread [ 64 ]; // максимальное количество потоков
для показа
int num = Thread.enumerate( threads );
for( int i = 0; i < num; i++ )
System.out.println( threads[i] +":"+ threads[i].getState() );
```

Вы, вероятно, не будете использовать это API в общем программировании, но это интересно и полезно для экспериментирования и изучения потоков Java.

Временные интервалы

В дополнение к работе с приоритетами все современные системы (за исключением некоторых встроенных сред и «микросред» Java) реализуют временные интервалы потока. В системе с временным интервалом обработка потока нарезана таким образом, что каждый поток выполняется короткий период времени перед тем, как контекст переключится на другой поток, как показано на рис. 9.5.

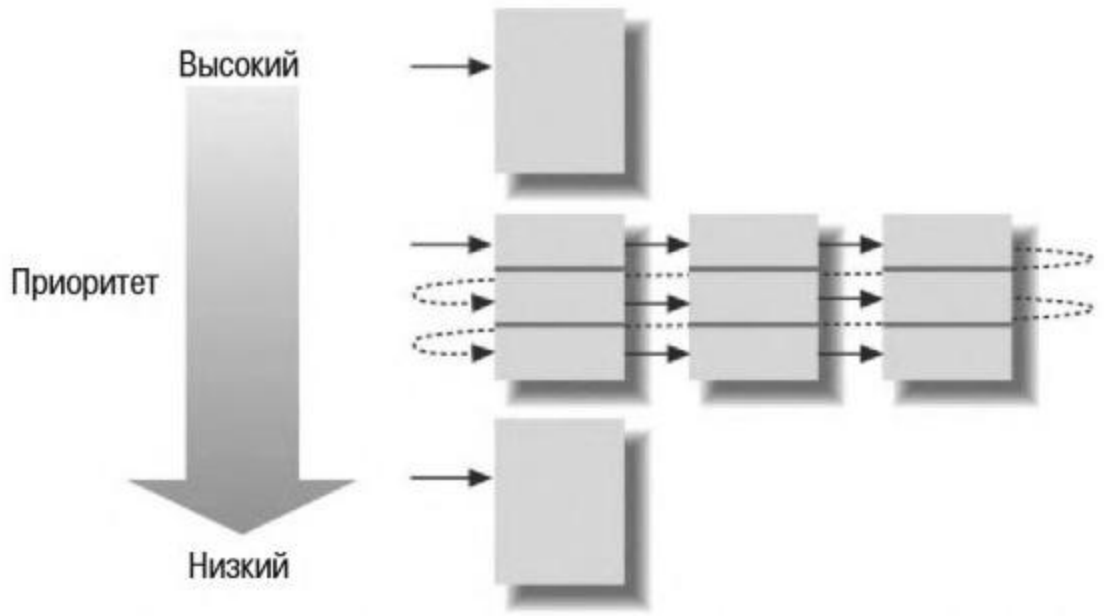

Рис. 9.5. Планирование с вытесняющим приоритетом и временными интервалами

Потоки с высоким приоритетом в этой схеме все еще вытесняют потоки с низким приоритетом. Добавление временных интервалов смешивает обработку среди потоков с одним и тем же приоритетом; на многопроцессорной машине потоки даже могут выполняться одновременно. Это может приводить к появлению разницы в поведении приложений, которые не используют потоки и синхронизацию должным образом.

Строго говоря, поскольку Java не гарантирует временных интервалов, вы не должны писать код, который полагается на этот тип планирования; любое программное обеспечение, которое вы пишете, должно функционировать под циклическим планированием. Если вы интересуетесь, что делает ваша разновидность Java, попробуйте следующий эксперимент:

```
public class Thready {
public static void main( String args [] ) {
new ShowThread("Foo").start();
new ShowThread("Bar").start();
}
static class ShowThread extends Thread {
String message;
```

```
ShowThread( String message ) {
this.message = message;
}
public void run() {
while ( true )
System.out.println( message );
}
}
}
```

Класс Thready пускает в ход два объекта ShowThread. ShowThread — это поток, который идет в постоянный цикл (очень плохая форма) и выводит его сообщение. Поскольку мы не указываем приоритет ни для одного из потоков, они оба наследуют приоритет их создателя, поэтому имеют одинаковый приоритет. Когда вы запустите этот пример, вы увидите, как ваша реализация Java совершает планирование. По циклической схеме должно быть выведено только сообщение Foo; Bar никогда не появляется. В реализации с временным интервалом вы иногда увидите чередование сообщений Foo и Bar.

Приоритеты

Как мы говорили ранее, приоритеты потоков существуют как общие указания относительно того, как реализация должна распределять время между соревнующимися потоками. К сожалению, учитывая сложность того, как потоки Java назначаются начальным реализациям потоков, вы не можете положиться на точное значение приоритетов. Вместо этого вы только должны рассматривать их как подсказку виртуальной машине.

Давайте поиграем с приоритетами наших потоков:

```
class Thready {
public static void main( String args [] ) {
Thread foo = new ShowThread("Foo");
foo.setPriority( Thread.MIN_PRIORITY );
Thread bar = new ShowThread("Bar");
bar.setPriority( Thread.MAX_PRIORITY );
bar.start();
}
}
```

Мы ожидаем, что с этим изменением в нашем классе `Thready` поток Bar полностью поглотится. Если вы запустите этот код в реализации Solaris в Java 5.0, как раз это и произойдет. Но это не сработает в Windows или более старых версиях Java. Подобным образом, если вы смените приоритеты значений на что-то иное, кроме `min` и `max`, вы можете вообще не заметить никакой разницы. Тонкости, относящиеся к приоритету и выполнению, относятся к тому, как потоки и приоритеты Java связаны с реальными потоками в операционной системе. По этой причине приоритеты потоков должны быть отложены до разработки систем и фреймворков.

Возвращение

Всегда, когда поток спит, ожидает или блокирует ввод/вывод, он выдает свой временной слот и другой поток включается в график. Пока вы не пишете методы, которые используют постоянные циклы, все потоки должны получать то, что им надлежит. Однако поток также может сигнализировать, что он хочет отдать свое время добровольно в любой момент при помощи вызова `yield()`. Мы можем изменить наш предыдущий пример, чтобы он включал `yield()` для каждой итерации.

```
...
static class ShowThread extends Thread {
...
public void run() {
while ( true ) {
System.out.println( message );
yield();
}
}
}
```

Вы должны видеть, что сообщения Foo и Bar строго сменяются. Если у вас есть потоки, которые выполняют очень интенсивные вычисления или еще каким-то образом пожирают время центрального процессора, вы можете найти подходящее место для них, чтобы возвращать контроль время от времени. Как вариант вы можете захотеть сбросить приоритет вашего потока, требующего большого объема вычислений, так что более важная обработка сможет продолжаться вокруг него.

К сожалению, спецификация языка Java является очень слабой по отношению к методу `yield()`. Это еще одна вещь, которую вы должны рассматривать как подсказку для оптимизации, а не гарантию. В худшем случае система выполнения может просто игнорировать вызов `yield()`.

Группы потоков

Класс `ThreadGroup` позволяет нам массово работать с потоками: мы можем использовать его, чтобы организовать потоки в группы и работать с группами в целом. Группа потоков может содержать другие группы потоков вдобавок к индивидуальным потокам, поэтому наша компоновка может быть иерархической. Группы потоков особенно полезны, когда мы хотим начать задачу, которая может сама создать много потоков. Назначая задаче группу потоков, позже мы можем идентифицировать все потоки задачи и управлять ими. Группы потоков являются также предметом ограничений, которые могут быть наложены диспетчером безопасности Java, поэтому мы можем ограничить поведение потока согласно его группе потока. Например, мы можем запретить потокам в отдельной группе взаимодействовать с потоками в другой группе. Это один из способов, которым веб-браузеры могут запретить потокам, запущенным апплетами Java, останавливать важные потоки системы.

Когда мы создаем поток, обычно он становится частью группы потоков, к которой принадлежит выполняемый в данный момент. Чтобы создать новую группу потоков, мы можем вызвать конструктор:

```
ThreadGroup
myTaskGroup = new ThreadGroup("Группа моей задачи");
```

Конструктор `ThreadGroup` берет имя, которое может использовать отладчик, чтобы помочь вам определить группу. (Вы также можете назначать имена самим потокам.) Когда у нас есть группа, мы можем помещать в нее потоки, поставляя объект `ThreadGroup` в качестве аргумента конструктору `Thread`:

```
Thread myTask = new Thread( myTaskGroup, taskPerformer );
```

Здесь `myTaskGroup` является группой потока, а `taskPerformer` — целевым объектом (объектом интерфейса `Runnable`, который выполняет задание). Любые дополнительные потоки, которые создает `myTask`, также принадлежат группе потока `myTaskGroup`.

Работа с классом `ThreadGroup`

Класс `ThreadGroup` существует, так что вы можете управлять потоками в группах. Он обладает методами, которые являются параллельными базовым методам управления `Thread` — даже устаревшим `stop()`, `suspend()` и `resume()`. Эти методы оперируют всеми потоками в группе потоков. Вы также можете пометить группу потока как «демоническую»; демоническая группа потоков автоматически убирается, когда пропадают все ее дети. Если группа потока не является демонической, вам придется вызвать `destroy()`, чтобы убрать ее, когда она опустеет.

Мы можем установить максимальный приоритет для потоков, созданных в группе потока, вызвав метод `setMaximumPriority()`. После этого в группе потоков не могут создаваться никакие потоки с приоритетами выше максимального; потоки, которые меняют свой приоритет, не могут установить его выше максимума.

В конце концов, вы можете получить список всех потоков в группе. Метод `activeCount()` говорит вам, как много потоков находится в группе; метод `enumerate()` дает вам их список. Мы использовали метод `enumerate()` раньше, когда показывали состояние всех потоков в дефолтной группе потоков, используя статический метод `Thread.enumerate()`. Аргументом `enumerate()` является массив классов `Thread`, который наполняет `enumerate()` группой потоков. `activeCount()` и `enumerate()` действуют рекурсивно на всех группах потока, которые содержатся в группе.

Непроверяемые исключения

В Java непроверяемые исключения, которые не обрабатываются никаким методом, в конце концов, всплывают в методе `run()` выполняющегося потока и выбрасываются оттуда. По умолчанию Java справляется с этим, просто занося их в поток системных ошибок или системный журнал и прекращая поток. Однако вы можете самостоятельно указать поведение «ловушки» для этих исключений, создав подклассы `ThreadGroup` и переопределив метод `uncaughtException()`. Когда необрабатываемое исключение генерируется, оно передается этому методу, который может применить некоторые действия или снова выбросить его прежде, чем поток завершится.

В Java 5.0 этот шаблон был расширен определением интерфейса `Thread.UncaughtExceptionHandler` и добавлением обработчиков необра-

ботанных исключений для потоков и системных исключений в дополнение к обработчику исключения для `ThreadGroup`. Мы можем обрабатывать необработанные исключения для одного потока таким образом:

```
Thread thread = new Thread();
thread.setUncaughtExceptionHandler(
new Thread.UncaughtExceptionHandler() {
public void uncaughtException( Thread t, Throwable e ) {
System.err.println( t + " threw exception: " + e );
}
} );
```

Этот пример выводит исключение до смерти потока. Мы могли бы установить такой же обработчик на `ThreadGroup` таким же образом или назначить его всем исключениям, используя статический метод `Thread.setDefaultUncaughtExceptionHandler()`.

Выполнение потока

То, как приложения используют потоки и связанные с ними затраты и выгоды, очень повлияло на разработку многих API Java. Мы обсудим некоторые из этих вопросов в подробностях в других главах. Но стоит кратко упомянуть некоторые аспекты выполнения потока и то, как использование потоков диктует форму и функционал некоторых последних пакетов Java.

Издержки синхронизации

Процесс получения блокировок для синхронизации потоков занимает время, даже если нет разногласий. В ранних реализациях Java это время могло быть значительным. С более новыми виртуальными машинами оно почти ничтожно. Однако ненужная синхронизация нижнего уровня может все же замедлить приложения, блокируя потоки, где в противном случае узаконенный параллельный доступ может быть разрешен. Из-за этого два важных API, Java Collections API и Swing GUI API были разработаны особым образом, чтобы избежать ненужной синхронизации, помещая ее под контроль разработчика.

Пакет `java.util` интерфейса Collections API замещает ранние, простые составные типы Java — а именно `Vector` и `Hashtable` — более об-

работанными и, в частности, несинхронизированными типами (`List` и `Map`). Вместо этого Collections API подчиняется коду приложения, чтобы синхронизировать доступ к библиотекам, когда необходимо, и предоставляет специальный функционал «списка ошибок», чтобы помочь обнаружить параллельный доступ и выбросить исключение. Она также предоставляет «обертки» синхронизации, которые могут дать безопасный доступ в старом стиле. Специальные дружелюбные реализации с параллельным доступом библиотек `Map` и `Queue` включены в пакет `java.util.concurrent`. Более того, эти реализации написаны так, чтобы позволить высокую степень параллельного доступа без какой-либо синхронизации пользователя. Мы поговорим о них в главе 11.

Java Swing GUI, которая выросла из AWT, использует другой подход к обеспечению скорости и безопасности. Swing диктует, что изменение ее компонентов (с заметными исключениями) должно быть полностью совершено единственным потоком: главной очередью событий. Swing решает проблемы выполнения, а также неприятные трудности предопределения в порядке событий, заставляя единственный суперпоток контролировать GUI. Приложение может обращаться к очереди событий косвенно, отправляя команды в очередь посредством простого интерфейса.

Потребление ресурсов потоком

Фундаментальным шаблоном в Java, который мы будем иллюстрировать в главах 12 и 13, является запуск множества потоков для обработки несинхронных внешних ресурсов, таких как соединения между сокетами. Для максимальной эффективности можно заставить веб-сервер создавать поток для каждого клиентского соединения, которое он обслуживает. Когда каждый клиент имеет собственный поток, операции ввода/вывода могут блокироваться и начинаться заново при необходимости. Но как бы это ни было эффективно в отношении производительности, это очень нерациональное использование ресурсов сервера. Потоки потребляют память; каждый поток имеет собственный «стек» для локальных переменных, и переключение между работающими потоками (контекстное переключение) увеличивает накладные расходы центрального процессора. В то время как потоки являются легковесными (в теории возможно иметь сотни или тысячи потоков, работающих на большом сервере), в определенный момент ресурсы, потребляемые самими потоками, начинают разрушать причину создания

большего количества потоков. Часто к этому приводят всего несколько дюжин потоков. Создание потока для клиента не всегда является масштабируемой опцией.

Альтернативным подходом является создание «пула потоков», где фиксированное количество потоков получает задания из очереди и возвращается, чтобы получить еще, когда задание выполнено. Это повторное использование потоков способствует твердой массштабируемости, но его исторически было сложно реализовать для серверов в Java, потому что поточный ввод/вывод (для таких вещей, как сокеты) полностью не поддерживал неблокирующие операции. Это изменилось с Java 1.4 и введением пакета NIO (новый ввод/вывод), `java.nio`. Пакет NIO представил асинхронные каналы ввода/вывода: неблокирующее чтение и запись плюс возможность «выбирать» или тестировать готовность потоков к перемещению данных. Каналы также могут быть асинхронно закрытыми, что позволяет потокам работать с ними достойно. С пакетом NIO возможно создавать серверы с более продвинутыми масштабируемыми шаблонами потоков.

В Java 5.0, пулы потоков и сервисы «выполнения» работы были систематизированы как утилиты как часть нового пакета `java.util.concurrent`, что означает, что вы не должны их писать сами. Мы поговорим о них позже, когда будем обсуждать утилиты согласованности в Java.

Средства для поддержки параллелизма

К настоящему времени в этой главе мы показали, как создавать и синхронизировать пакеты на низком уровне, используя простые типы языка Java. Пакет `java.util.concurrent` и субпакеты, представленные в Java 5.0, совершенствуют этот функционал, добавляя важные утилиты потоков и систематизируя некоторые распространенные шаблоны дизайна, поставляя стандартные реализации. Грубо говоря, для универсальности эти области включают:

Реализации библиотек, поддерживающих потоки

Пакет `java.util.concurrent` прибавляет к Java Collections API несколько реализаций специфических моделей потоков. Они включают ожидание с ограничением во времени и блокирующие реализации интерфейса `Queue`, а также неблокирующие оптимизированные реализа-

ции интерфейсов Queue и Map с параллельным доступом. Пакет также добавляет реализации List и Set с «копированием при записи» для крайне эффективных «почти всегда читаемых» случаев. Это может звучать сложно, но в действительности очень хорошо охватывает некоторые достаточно простые случаи. Мы рассмотрим Collections API в главе 11.

Executors

Классы Executors выполняют задания, включая интерфейсы Runnable, и абстрагируют концепцию создания потока и объединения в пул пользователем. Исполнители призваны быть высокоуровневой заменой идиомы создания новых потоков, чтобы служить серии заданий. Вместе с интерфейсами Executor был представлен интерфейс Callable и класс Future, которые расширяются над Runnable, чтобы позволить управление, возвращение значений и обработку исключений.

Конструкции синхронизации низкого уровня

Пакет java.util.concurrent.locks содержит набор классов, включая Lock и Condition, который является параллельным синхронизации простых типов на языковом уровне Java и продвигает их до уровня конкретного API. Пакет блокировок также добавляет концепцию неразделительных блокировок чтения/записи, позволяющих большую параллельность в синхронизированном доступе к данным.

Конструкции синхронизации высокого уровня

Включают классы CyclicBarrier, CountDownLatch, Semaphore и Exchanger. Эти классы реализуют общие шаблоны синхронизации, срисованные из других языков и систем, и могут служить базой новых высокоуровневых инструментов.

Атомарные операции (звучит очень в стиле Джеймса Бонда, не так ли?)

Пакет java.util.concurrent.atomic предоставляет обертки и утилиты для атомарных операций «все-или-ничего» над простыми типами и ссылками. Это включает простые комбинации атомарных операций, как тестирование значения перед его установлением и получение и увеличение числа в одной операции. За возможным исключением оптимизаций, производимых виртуальной машиной Java для пакета операций atomic, все эти утилиты реализуются в чистом Java на вершине стандартных конструкций синхронизации языка Java. Это означает, что они

в каком-то смысле являются только удобными утилитами и в действительности не добавляют новых возможностей языку. Их главная роль заключается в том, чтобы предлагать стандартные шаблоны и идиомы в потоках Java и делать их безопаснее и эффективнее в использовании. Хорошим примером этого является утилита `Executor`, которая позволяет пользователю управлять набором задач в предопределенной модели потоковой обработки без углубления в создание потоков. Такие высокоуровневые API, как эти, одновременно упрощают кодирование и позволяют улучшенную оптимизацию общих случаев. Далее в этой главе мы рассмотрим каждую из этих тем, за исключением реализаций библиотек. Мы обсудим их, когда перейдем к Java Collections API в главе 11. Но для начала мы должны выразить благодарность Дугу Ли, автору книги Concurrent Programming in Java (Addison-Wesley, 1999), который руководил группой, добавившей эти пакеты в Java и в большой степени ответственной за их создание.

Исполнители

В этой главе мы создали множество классов `Thread` и надеемся, что показали, как использовать их эффективно. Но по большому счету потоки являются довольно низкоуровневым инструментом программирования и, если о них не заботиться, могут быть подвержены ошибкам. Когда мы распознаем определенные общие шаблоны, которые снова и снова создают разработчики, используя потоки, естественно желать повысить шаблон до уровня API. Одна такая связанная пара шаблонов — это концепция услуги исполнения, которая управляет задачами, и шаблона пула потоков, который служит задачам эффективным способом. Огромное количество разработчиков годами снова и снова реализовывали пулы потоков тем или иным способом, и если вы добавляете такие свойства, как планирование различных моделей потоков, они могут становиться довольно сложными. Чтобы решать такие проблемы, пакет `java.util.concurrent` включает интерфейсы для многих дефолтных реализаций шаблона исполнителя для общих моделей потоков. Это включает сложное планирование, а также асинхронную библиотеку результатов заданий, если они этого требуют. В общем, вы можете использовать `Executor` как замену создания одноразовых потоков везде, где вам нужно выполнить объекты интерфейса `Runnable`. Преимуществом является то, что понимание и последующее изменение поведения вашего кода будут намного легче, когда вы работаете на этом уровне.

Для простого случая выполнения ряда задач и наблюдения за их выполнением мы можем рассматривать базовый интерфейс `Executor`, который выполняет объекты интерфейса `Runnable`. Удобство `Executor` заключается в том, что его компаньон класс утилиты `Executors` является фабрикой для создания разных типов реализаций `Executor`. Мы поговорим о различных типах, которые он может создать вскоре, но сейчас давайте использовать метод, называемый `newFixedThreadPool()`, который, как подсказывает его имя, возвращает `Executor`, реализованный при использовании пула потоков определенного размера:

```
Executor executor = Executors.newFixedThreadPool( 3 ) ; // 3 потока
List<Runnable> runnables = ... ;
for( Runnable task : runnables )
executor.execute( task );
```

Здесь мы предлагаем ряд задач `Runnable` нашему `Executor`, который выполняет их, используя пул с максимум тремя потоками. Если наш список содержит более трех задач, тогда некоторым из них придется ждать, пока поток будет свободен, чтобы обслужить их. Итак, что же происходит, когда мы представляем четвертый элемент? Интерфейс `Executor` в действительности не указывает этого. Решает отдельная реализация. Не уточняя больше ничего об этом типе, мы не знаем, собирается ли `Executor` поставить в очередь задачи или он будет использовать пул, чтобы обслужить их? Некоторые реализации `Executor` могут блокировать или даже выполнять `Runnable` сразу по вызову `execute()` в потоке вызывающего оператора. Но в этом случае (и для всех реализаций `Executor`, созданных для нас методами фабрики класса `Executors`), задачи в сущности определены в неограниченную очередь. В примере наш цикл предлагает все задачи немедленно, и они поставлены в очередь исполнителем, пока три потока не обслужат их. С помощью всего лишь одной или двух строчек кода в нашем примере мы можем помешать параллельности нашего списка задач и избежать подробностей создания каких-либо потоков самостоятельно. Позже, если мы решим, что мы лучше будем выполнять все задачи по одной в единицу времени, изменение будет тривиальным (запуск только одного потока!). Далее мы поднимемся еще на одну ступеньку и посмотрим на выполнимые задачи, которые производят значения, и исполнителей, которые могут планировать задачи за нас.

Задачи с результатами: `Callable` и `Future`

Поскольку интерфейс `Runnable` создан для потребления `Threads`, его интерфейс API не дает прямого ответа вызывающему оператору. Но-

вый интерфейс Callable, являющийся эффективным заменителем для Runnable, исправляет эту ситуацию, предоставляя метод call(), который возвращает результат и может выбросить исключение. Callable является общим классом, который параметризируется возвращаемым им типом. В следующих примерах создается интерфейс Callable, который возвращает целое число:

```
class MyCallable implements Callable<Integer> {
public Integer call() { return 2+2; }
}
// или анонимно
Callable<Integer> callable = new Callable<Integer>() {
public Integer call() { return 2+2; }
};
```

Существует также удобный метод соединения интерфейсов Runnable и Callable в классе Executors. Он берет Runnable и фиксированное значение, чтобы вернуть в качестве значения, когда он выполняется:

```
Callable<Integer> callable = Executors.callable( runnable,
42 /*возвращаемое значение*/ );
```

Новый класс Future используется с Callable и служит удобным средством для ожидания и возвращения результата задачи или отмены задачи перед ее выполнением. Future возвращается методами submit() класса ExecutorService, который изначально является усовершенствованным Executor.

Мы обсудим ExecutorServices в следующей главе.

```
Future<Integer> result = executorService.submit( callable );
int val = result.get(); // блокируется до готовности
```

Future также является общим интерфейсом, который параметризируется его возвращаемым типом. Это объясняет немного претенциозное имя. Например, Future<Integer> может быть прочтен как «будущее целое число». Интерфейс Future обладает блокирующим методом и методом ожидания get(), чтобы получить результат, когда он готов, а также тестовым методом isDone() и методом cancel(), чтобы остановить задачу, если она еще не началась. Если задача была отменена, вы получаете исключение CancellationException, когда пытаетесь получить результат. Итак, мы достаточно рассказали об этих интерфейсах. Далее мы посмотрим на ExecutorService, который их использует.

Интерфейс ExecutorService

Наш первый интерфейс `Executor` был немного более чем воронкой для интерфейсов `Runnable` и, как мы описывали, требовал знаний реализации, чтобы понимать, как он станет справляться с задачами. В отличие от него интерфейс `ExecutorService` предназначен для того, чтобы быть асинхронизированным обработчиком задач. Вместо метода `execute()` он обладает методами `submit()`, принимающими `Callable` (или `Runnable`) и возвращающимися сразу же с объектом `Future`, который может использоваться для управления задачами и сбора результатов позже. В дополнение к этому `ExecutorService` имеет жизненный цикл, определенный его собственным методом `shutdown()` и связанными методами, которые могут быть использованы для остановки сервиса после того, как задачи выполнены.

`ExecutorService` расширяет интерфейс `Executor`. На самом деле все реализации, возвращенные методами генерации `Executor`, являются `ExecutorServices` — включая тот, который мы использовали в нашем первом примере. Мы посмотрим на эти методы генерации, чтобы увидеть, какие виды услуг предлагаются.

Класс `Executors` предлагает три типа реализаций `ExecutorService`.

`newFixedThreadPool(int)`

Это классический пул потоков с указанным максимальным размером пула и неограниченной очередью для передачи задачи. Если поток погибает по какой-то причине во время выполнения задачи, для его замещения будет создан новый поток. Потоки никогда не убираются из пула, пока сервис не закроется.

`newCachedThreadPool()`

Этот пул использует неограниченное количество потоков, которое растет и сокращается по требованию. Главным преимуществом этого сервиса является то, что потоки кешируются на определенный период и используются заново, устраняя ограничение создания новых потоков для кратковременных задач. Потоки, которые не используются одну минуту, устраняются. Задачи предоставляются непосредственно потокам; реальной очереди не существует.

`newSingleThreadExecutor()`

Этот интерфейс `ExecutorService` использует единственный поток для выполнения задач из неограниченной очереди. В этом смысле он идентичен фиксированному пулу потоков с размером пула равным 1.

Давайте посмотрим на более реалистичное использование `ExecutorService`, выведенное из примера `TinyHttpd` в главе 13. В этой главе мы создаем мини-веб-сервер, чтобы проиллюстрировать свойства сетевых API. Здесь мы не будем показывать сетевые детали, но реализуем главный цикл отправки запросов для примера, используя выполняющий сервис пула потоков. (Перейдите к главе 13, чтобы увидеть реализацию обработчика класса клиентского соединения `Runnable`. Класс работает одинаково хорошо в обоих примерах.) Поехали:

```
public class ExecutorHttpd
{
ExecutorService executor = Executors.newFixedThreadPool(3);
public void start( int port ) throws IOException
{
final ServerSocket ss = new ServerSocket( port );
while ( !executor.isShutdown() )
executor.submit( new TinyHttpdConnection( ss.accept() ) );
}
public void shutdown() throws InterruptedException {
executor.shutdown();
executor.awaitTermination( 30, TimeUnit.SECONDS );
executor.shutdownNow();
}
public static void main( String argv[] ) throws Exception
{
new ExecutorHttpd().start( Integer.parseInt(argv[0]) );
}
}
```

Класс `ExecutorHttpd` содержит экземпляр фиксированного пула потоков `ExecutorService` с тремя потоками для обслуживания клиентских соединений. В методе `start()` нашего класса мы создаем `ServerSocket`, который принимает входящие сетевые соединения. Затем мы входим в цикл, который выполняется, пока наш сервис не будет помечен для закрытия. Внутри цикла мы создаем новый обработчик соединения (экземпляр интерфейса `Runnable` класса `TinyHttpdConnection`) для каждого соединения и передаем его исполнителю. Метод нашего класса `shutdown()` иллюстрирует достойное завершение. Во-первых, мы вызываем метод `shutdown()` для нашего исполнителя, который заставляет сервис прекратить принимать новые задачи и позволяет завершиться выполняющимся в данный момент. Затем мы ждем приемлемый период

времени, пока завершатся все интернет-запросы (30 секунд), используя метод `awaitTermination()` перед попыткой менее достойного завершения при помощи `shutdownNow()`. `shutdownNow()` пытается прервать или другим способом остановить потоки как можно быстрее. Мы оставляем все как есть, но метод в действительности возвращает класс `List` задач, которые остаются подвешенными после попытки. В конце концов у нас есть метод `main()`, который тренирует наш пример, создавая экземпляр `ExecutorHttpd` на порту, указанному как аргумент программы.

Совокупные задачи

В дополнение к методам `submit()` своих индивидуальных задач `ExecutorService` также предлагает набор совокупных методов выполнения `invokeAll()` и `invokeAny()`, которые представляют множество задач как группу и возвращают результаты, когда они выполнены или когда выполнится первый, соответственно. Представим наш первый пример, используя класс `List` интерфейсов `Callable`, таким образом:

```
List<Callable<Integer>> taskList = ...;
ExecutorService execService = Executors.newFixedThreadPool(3);
List<Future<Integer>> resultList = execService.invokeAll( taskList );
```

В отличие от этого метод `invokeAny()` возвращает только первый успешно выполненный результат задачи (отменяя все оставшиеся невыполненные задачи):

```
int result = execService.invokeAny( taskList );
```

Оба метода также предлагают версии с ожиданием с ограничением во времени, которые прерываются после указанного периода времени.

Спланированные задачи

Для задач, которые вы хотите запустить в будущем или запускать с определенной периодичностью, используйте интерфейс `Scheduled ExecutorService`. `ScheduledExecutorService` является интерфейсом `ExecutorService` с дополнительными методами «планирования», которые задерживаются для выполнения интерфейсов `Runnable` или `Callable` или период спецификации для `Runnable`. Два дополнительных метода генерации класса `Executors` производят спланированных исполнителей:

```
Executors.newScheduledThreadPool(int);
Executors.newSingleThreadScheduledExecutor();
```

Это то же самое, что и одинаково названные методы для сервисов регулярных выражений за исключением возвращения запланированного выполняемого типа.

Чтобы выполнить задачу в будущем, вы указываете задержку от текущего времени. Например:

```
ScheduledExecutorService exec = Executors.newScheduledThreadPool(3);
exec.schedule( runnable, 60, TimeUnit.SECONDS ); // выполнять одну минуту в
// будущем
// выполнять в указанную дату и время
Calendar futureDate = ...; // конвертировать из календаря
Date date = futureDate.getTime(); // в Date
long delay = date.getTime() - System.currentTimeMillis(); // в связанные
// миллисекунды
exec.schedule( runnable, delay, TimeUnit.MILLISECONDS ); // запускать
в указанную
// дату
```

Для временной работы существует два типа повторяющихся графиков — фиксированная задержка и фиксированный объем. *Фиксированная задержка* означает, что между окончанием выполнения задачи и началом следующего выполнения проходит фиксированное количество времени. *Фиксированный объем* означает, что задача должна начинать выполнение через фиксированные интервалы времени, независимо от того, как много времени она занимает. Разница играет роль, когда время на выполнение задачи долгое по сравнению с интервалом. Следующий отрывок планирует очистку журнальных файлов, которая начнется через 12 часов, и после этого будет происходить каждые 12 часов:

```
Runnable cleanup = new Runnable() {
public void run() { cleanUpLogFiles(); }
};
long period = 12*60*60, delay = period; // секунды
Future<?> logService = executionService.scheduleAtFixedRate(
cleanup, delay, period, TimeUnit.SECONDS );
```

Поскольку задача для периодического планирования является элементом интерфейса `Runnable`, объект Future не возвращает полезное значение (он возвращает `null`), поэтому мы не указываем тип параметра в его реализации обобщенного типа. Интерфейс `Future` также позволяет при желании отменить задачу впоследствии:

```
logService.cancel();
```

Мы должны упомянуть, что интерфейс ScheduledExecutorService имеет значительное сходство с классом java.util.Timer, который мы обсудим в главе 11, особенно с учетом периодических графиков. Однако java.util.Timer всегда однопоточен.

Интерфейс CompletionService

CompletionService является легковесным внешним интерфейсом типа очереди для исполнителя. CompletionService предоставляет методы submit(), которые делегируют их задачи определенному экземпляру Executor, а затем предоставляет методы take() и poll() для получения результатов Future для выполненных задач. Считайте CompletionService нянькой для Futures, которая позволяет вам легко собирать только завершенные результаты (в отличие от самостоятельной проверки Future, чтобы увидеть, какие завершились и в каком порядке). ExecutorCompletionService является конкретной реализацией CompletionService, которая принимает Executor в свой конструктор:

```
Executor executor = Executors.newFixedThreadPool(3);
CompletionService<Integer> completionService =
new ExecutorCompletionService<Integer>( executor );
completionService.submit( callable );
completionService.submit( runnable, resultValue );
// получение результата
Future<Integer> result = completionService.poll();
if ( result != null )
// использовать значение…
// блокировка, ожидание результата
Future<Integer> result = completionService.take();
```

Реализация ThreadPoolExecutor

В этой главе мы обращались к различным сервисам исполнителей, созданных генераторами Executors как различные реализации интерфейса ExecutorService. Но эти реализации являются лишь разными конфигурациями одной очень гибкой реализации ExecutorService, называемой ThreadPoolExecutorService. Вы можете использовать это реализацию непосредственно, если хотите; она предлагает некоторые дополнительные свойства. Первоначальный конструктор ThreadPoolExecutorService позволяет вам указывать размер пула «основного» потока и максимальный размер, а также значение тайм-аута потока для удаления бездей-

ствующих потоков. Основная величина — это минимальное количество потоков, которому после создания разрешается существовать бессрочно. Конструктор позволяет вам предоставлять очередь задач (реализацию интерфейса `BlockingQueue`), в которую помещены новые задачи. Это последнее свойство позволяет вам управлять политикой очередей самостоятельно. Вы можете указать очередь с ограниченной вместимостью:

```
ExecutorService executorService = new ThreadPoolExecutor(
corePoolSize, maximumPoolSize, keepAliveTime, timeUnit, taskQueue );
```

Реализация `ThreadPoolExecutor` также обладает методами, которые позволяют вам менять базовый и максимальный размер пула, пока сервис является активным, или «предзапуск» базовых потоков перед использованием сервиса.

В действительности эти последние свойства поднимают очень интересный вопрос. Если мы знаем, что наш выполняющий сервис является реализацией `ThreadPoolExecutor`, мы можем привести ее во время выполнения, чтобы получить доступ к этим дополнительным методам и к таким вещам, как изменение размера пула. Может, это и не то, что разработчики некоторых сервисов имели в виду; в действительности это может быть откровенно опасно не в тех руках. По этой причине класс `Executors` предлагает ряд «ненастраиваемых» методов оберток, ведущих себя как «неизменяемые» методы библиотек, которые мы увидим в Java Collections API. Эти методы оборачивают выполняющий сервис в делегирующем объекте, который не демонстрирует реализацию вызывающему оператору:

```
ExecutorService tweakable = Executors.newFixedThreadPool();
ExecutorService safe = Executors.unconfigurableExecutorService( tweakable );
```

Сервер приложения может, например, обернуть сервис, чтобы защитить себя от индивидуальных приложений, изменяющих (намеренно или случайно) общий сервис, используемый многими приложениями.

Создание потоков

Мы сказали, что шаблон `Executor` является общей заменой использованию классов `Thread` для выполнения простых задач. Хотя классы `Executor` защищают нас от создания `Thread`, в некоторых случаях мы все же можем хотеть иметь контроль над тем, как конструируются или настраиваются потоки, используемые в различных реализациях пулов потоков. Для этой цели и стандартизации создания `Thread` в общем пакет параллелизма добавляет явную генерирующую API для создания потоков.

Интерфейс `ThreadFactory` предоставляет метод `newThread()`. Одна из этих фабрик используется всеми реализациями сервисов, которые создают потоки. Все методы генерации классов `Executor` имеют дополнительную форму, которая принимает явный класс `ThreadFactory` в качестве аргумента. Вы можете получить дефолтную фабрику потоков, используемую ими, при помощи метода `Executors.defaultThreadFactory()`. Вы можете предоставить собственный `ThreadFactory`, чтобы выполнить пользовательскую настройку, такую как переменные или приоритеты `ThreadLocal`.

Фреймворк `Fork/Join`

До этого момента мы видели, как утилиты параллельности Java могут использоваться для управления простыми сценариями параллельного программирования. Мы видели, что можем запустить много задач в интерфейсы `ExecutorService` и собрать значения результатов, если необходимо, через интерфейсы `Future`. Мы видели, что можем планировать задачи, чтобы выполнять их в указанное время и с указанной частотой. Мы видели, что можем погружаться в подробности создания пула и контролировать степень параллелизма (то, как много потоков используется), если захотим. Позже в этой главе мы изучим API, которые помогают нам координировать потоки таким образом, что мы можем выполнять больше сложной работы, которая требует кооперации или явных фаз выполнения в их обработке данных. В этой главе мы рассмотрим API, которая помогает координировать задачи иным образом — помогая вам брать «масштабируемые» задачи и делить их, чтобы они соответствовали производительности, достижимой в любое время.

Давайте представим, что у вас есть задача, которая выполняет сложное вычисление вроде отрисовки видео или создания сложного изображения. Естественнее всего будет начать параллелизовать его с разделения работы для одного фрейма или изображения в фиксированное количество частей и подачи их в сервис исполнителя. Сервис исполнителя будет переключен, чтобы иметь столько потоков, сколько вы захотите использовать (возможно, количество, равное числу центральных процессоров или «ядер» на вашей машине), и назначит каждой части собственный поток. Если каждая задача (каждый кусок изображения) требует выполнения приблизительно одинакового количества работы и ничто другое не борется за время на вашем компьютере, тогда сценарий довольно оптимален. Мы ожидаем, что каждая часть изображения будет закончена приблизительно в одно время, и мы сможем сшить их

вместе эффективно. Но что если некоторые части изображения существенно сложнее отрисовать, чем другие? Что если одна часть использует в десять, или в сотню, или в тысячу раз больше мощности ЦПУ, чем другая? (Представьте, насколько быстрее может быть отрисовка пустой части изображения, например.) Тогда мы можем очутиться в ситуации, где многие потоки бездействуют, в то время как несколько потоков смешиваются, выполняя всю тяжелую работу. Что мы можем сделать, чтобы справиться с этим?

Что ж, первый вариант — более тщательно раздробить задачи. Мы можем сделать наши индивидуальные задачи настолько мелкими, что ни одна из них не сможет монополизировать поток на долгое время. Однако, когда задачи могут варьироваться в зависимости от степени сложности любой величины, это может привести к созданию очень большого количества мелких задач и, возможно, будет очень неэффективным — потокам придется переключаться между задачами и, что еще хуже, перемещать данные, чтобы обеспечить в некотором роде случайный порядок обработки заявок на выполнение работы. Нам необходим способ держать каждую задачу занятой, но позволять другим задачам помогать в случае перегруженности. В идеале мы также хотели бы минимизировать перебои в рабочем процессе в задачах, требующих переработки большого количества данных, чтобы избежать передачи потокам задач, которые требуют абсолютно новых данных, загружаемых в свою очередь.

Фреймворк `Fork/Join` является добавленной в Java 7 новой API, которая как раз предоставляет способ структурировать ваши задачи, так что они могут быть разделены при необходимости, чтобы оставлять все доступные потоки занятыми работой с данными настолько непрерывно, насколько это возможно. Точнее говоря, фреймворк `Fork/Join` полагается на задачи, которые могут быть разделены рекурсивно (то есть на две или более части с последующим разделением этих частей при необходимости и т. д.). Когда задача считается слишком большой, чтобы ее можно было выполнить быстро, она просто разделяется, и куски (теперь меньшие) помещаются в очередь для текущего потока. Затем фреймворк реализует то, что известно как алгоритм «перехвата работы», позволяя свободным потокам захватывать неначатые задачи у соседних очередей. Комбинация этих техник имеет серьезные преимущества. Во-первых, она позволяет избежать ненужной рандомизации загрузки. Потоки будут приступать к работе с непрерывными частями своих задач, пока они остаются занятыми. Для задач, требующих переработки большого количества данных, это может означать меньшую загрузку данных в потоках.

Однако при необходимости поток может забирать задачу у своего соседа. И тут возникает второе преимущество: по природе рекурсивного разделения задач самые большие / наименее раздробленные на куски задачи находятся внизу каждой очереди потока, как раз там, где соседний поток будет искать работу, чтобы украсть, если необходимо. Это означает, что когда работа схвачена, она будет перераспределена на максимально большие порции, стабилизирующие загрузку на поток, уменьшая похищение операций и переключения контекста. Это очень умный алгоритм, который берет начало в языке программирования `Cilk`.

Чтобы показать фреймворк `Fork/Join` во всей красе, мы произведем отрисовку изображения, которую используем как повод, чтобы нарисовать несколько фракталов! Фракталы являются удивительными математическими формами, которые возникают из относительно простых повторяющихся процессов. Тот, который мы будем рисовать сейчас, называется набором Мандельброта. Наш пример кода с Мандельбротом произведет свое рисование, используя фреймворк вилочного соединения, чтобы разделить работу рисования изображения между доступным количеством процессоров и держать их занятыми (рис. 9.6). Прежде чем мы начнем, несколько оговорок. Во-первых, мы не дадим подробных пояснений к рисующей части, которая высчитывает фрактал. Чтобы сфокусировать пример на фреймворке, мы сжали этот код до нескольких строк, которые выглядят очень загадочно. Пожалуйста, загляните в сноски, чтобы увидеть ссылку на разъяснение его работы. Дальше наш пример слепо раздробит куски изображения, пока они не достигнут фиксированного минимального размера. В то время как между потоками в этом случае произойдет перераспределение работы, лучший алгоритм, основанный на фактическом выполнении отрисовки и уменьшении ограничений ненужного распределения, может определить, в какой момент необходимо распределить работу. (У нас не будет большого количества данных, управляющих отрисовкой, и, таким образом, мы в основном фокусируемся на поддержании деятельности потоков, а не минимизации разделения.)

API фреймворка `Fork/Join` концентрируется на `ForkJoinPool` и различных реализациях подобия интерфейса `Future`, называемого `ForkJoinTask`. Фреймворк вилочного соединения можно использовать самыми разными способами — все зависит от того, как вы хотите структурировать задачи и что вы решите по поводу их деления (ответвления) и сбора результатов (соединение); однако мы только собираемся посмотреть на один распространенный случай. Мы будем использовать

подобие `ForkJoinTask`, называемое `RecursiveAction`, по сути являющееся `ForkJoinTask`, не возвращающим значение.

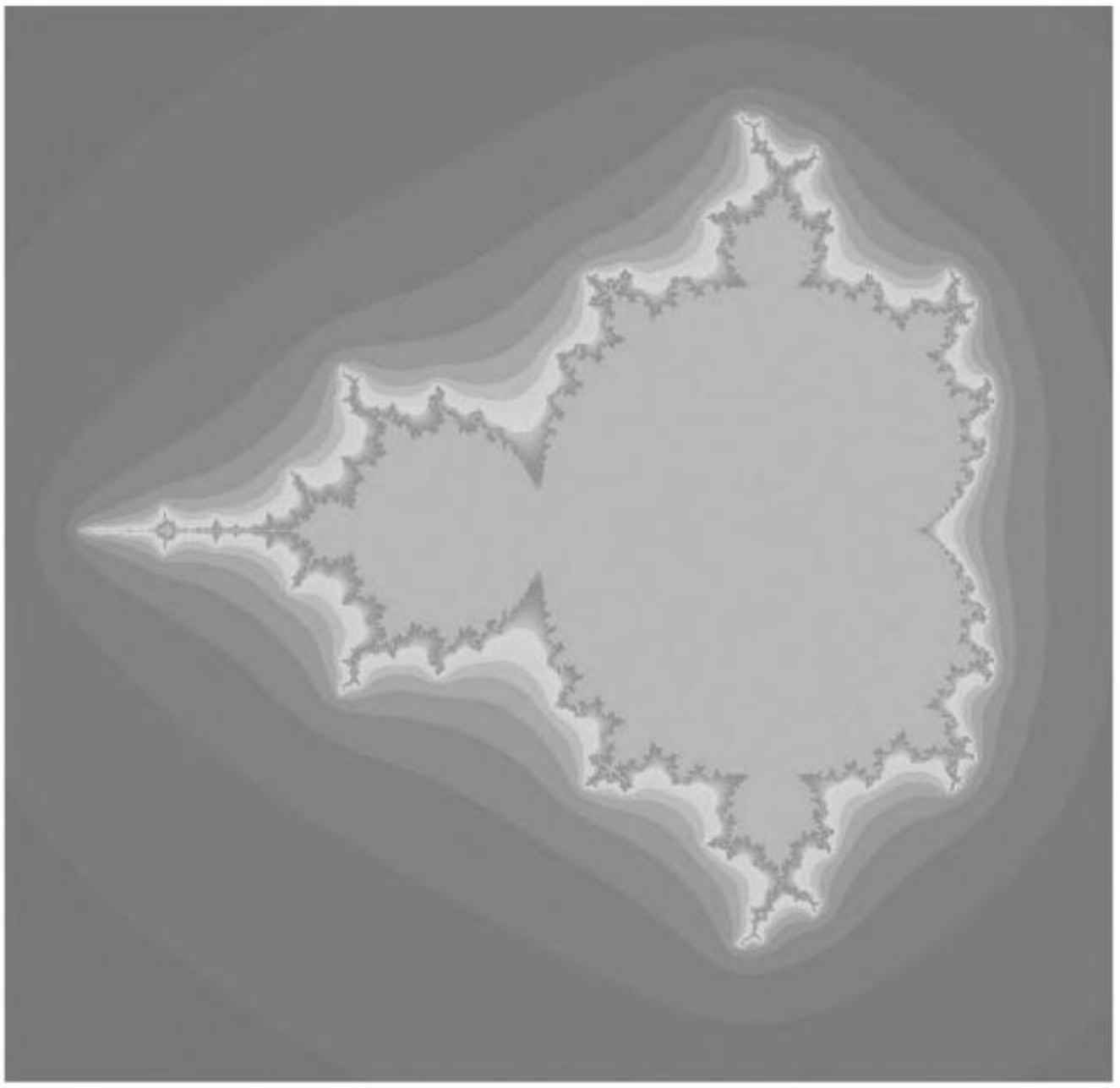

Рис. 9.6. Фреймоворк `Fork/Join` Мандельброта

Мы создадим подклассы `RecursiveAction` при помощи нашего `MandelbrotTask` и реализуем один необходимый абстрактный метод: `compute()`. Внутри метода `compute()` мы просто решим, разделять ли задачу или делать работу немедленно. Ниже представлен код:

```
public class Mandelbrot extends JFrame
{
@Override public void paint( Graphics g ) {
BufferedImage image = new BufferedImage( getWidth(), getHeight(),
BufferedImage.TYPE_INT_RGB );
ForkJoinPool pool = new ForkJoinPool(); // defaults thread per processor
pool.invoke( new MandelbrotTask( image, 0, image.getWidth()-1, 0,
image.getHeight()-1 ) );
g.drawImage( image, 0, 0, null );
}
```

```
public static void main( String[] args ) {
Mandelbrot mandy = new Mandelbrot();
mandy.setSize( 768, 768 );
mandy.setVisible( true );
}
}
class MandelbrotTask extends RecursiveAction
{
private static double size = 3.0, offsetX = -0.7, thresholdSq = 100;
private static int maxIterations = 30;
private BufferedImage image;
private int xStart, xEnd, yStart, yEnd;
private static int taskSplitThreshold = 1024;
MandelbrotTask( BufferedImage image, int xStart, int xEnd, int yStart,
int yEnd ) {
this.image = image;
this.xStart = xStart;
this.xEnd = xEnd;
this.yStart = yStart;
this.yEnd = yEnd;
}
public void render()
{
for ( int x = xStart; x <= xEnd; x++ ) {
for ( int y = yStart; y <= yEnd; y++ ) {
double r = x * size / image.getWidth() -size/2 + offsetX;
double i = y * size / image.getHeight() -size/2;
double zr=0, zi=0;
int iter;
for ( iter = 0; iter < maxIterations; iter++ ) {
double nzr = zr*zr - zi*zi + r;
double nzi = 2*zr*zi + i;
if ( nzr*nzr + nzi*nzi > thresholdSq ) { break; }
zr = nzr; zi=nzi;
}
image.setRGB( x, y, Color.HSBtoRGB( 0.5f * iter / maxIterations,
1.0f, 1.0f) );
}
}
}
@Override protected void compute()
```

```
{
int width = xEnd-xStart, height = yEnd-yStart;
if ( width*height < taskSplitThreshold ) {
render();
} else {
invokeAll(
new MandelbrotTask( image, xStart, xStart+width/2, yStart,
yStart+height/2 ),
new MandelbrotTask( image, xStart+width/2+1, xEnd, yStart,
yStart+height/2 ),
new MandelbrotTask( image, xStart, xStart+width/2,
yStart+height/2+1, yEnd ),
new MandelbrotTask( image, xStart+width/2+1, xEnd,
yStart+height/2+1, yEnd )
);
}
}
}
```

Попробуйте запустить пример, а затем поизменять размеры окна. Смотрите, как он перерисовывается, когда окно растягивается до больших размеров. Фрактал генерируется при рассмотрении каждой точки изображения как сложного числа (двумерное число) и повторяющемся применении к нему простой формулы: $Z=Z^2+C$, где Z — это изначально ноль, а C относится к координате точки. Затем мы окрашиваем точку в кодовый цвет, основываясь на том, как быстро это значение растет. В некоторых областях изображения значения растут быстро, и мы прекратим повторения на них; в других областях мы будем продолжать, пока не достигнем количества итераций (`maxIterations`). Это означает, что генерация некоторых частей займет больше времени, чем генерация других, и некоторые потоки, следовательно, будут еще перехватывать работу у других.

Метод `main()` класса `Mandelbrot` создает главное окно, `JFrame`. (Мы видели простой пример программирования GUI во введении этой книги и вернемся к нему в главе 16, когда будем говорить о Swing.) Главное, что мы должны знать здесь, это то, что метод `paint()` класса `JFrame` показывает изображение, находящееся в буфере, и наши разнообразные `MandelbrotTasks` соревнуются за отрисовку маленьких прямоугольников этого изображения.

Когда вызывается метод `paint()`, чтобы перерисовать экран, он создает новый `ForkJoinPool` и конструирует один `MandelbrotTask`.

`MandelbrotTask` заключает в себе знания об области изображения для рисования — изначально все изображение — и содержит методы для его отрисовки. Изначальный `MandelbrotTask` передается методу `invoke()` класса `ForkJoinPool`, являющемуся блокирующей формой метода `submit`, который будет ожидать, пока задача выполнится, перед тем как вернуться. Метод `paint()` затем нарисует полностью отрисованное изображение. Как вы видите, с точки зрения метода `paint()`, он предписывает одну задачу всему изображению и просто просит `ForkJoinPool` вызвать ее. Все рекурсивное разделение труда обрабатывается задачей при взаимодействии с фреймворком.

В методе `compute()` класса `MandelbrotTask` мы проверяем, сколько пикселов требуется отрисовать задаче. Если количество пикселов превышает указанный порог, мы разделяем область на четыре квадранта и создаем новый `MandelbrotTask` для каждого из них. Четыре подзадачи затем передаются унаследованному методу `invokeAll()`, который выполняет их и ожидает, пока они все выполнятся перед продолжением (он эффективно осуществляет над ними операцию соединения). Если количество пикселов ниже порогового, метод `compute()` прямо вызывает метод `render()`, чтобы создать фрактал для этой маленькой части изображения.

В нашем случае разделение задач продолжится, пока порог не будет достигнут и каждый из потоков в пуле будет занят отрисовкой частей экрана. Затем дерево задач свернется с каждым разделенным на подклассы `MandelbrotTask`, возвращающимся от вызова его метода `invokeAll()`, пока изначальная высокоуровневая задача не выполнится. Прежде чем мы продолжим: еще одно упражнение для тех, кого действительно интересует эта тема. Если вы захотите визуализировать то, какие потоки отрисовывают какие части, вы можете сделать следующее исключительно как эксперимент: внутри метода `render()` посмотрите на имя выполняющегося в данный момент потока с методом `ThreadgetName()`. Хотя это имя не будет значимым, оно будет уникальным для потока. Используйте `HashMap`, чтобы назначить имени уникальный номер и обозначить его этим номером каждый раз, когда вы его видите. Затем используйте этот номер, чтобы определить цвет отрисованного пиксела вместо логики фрактала (или объедините их, чтобы добавить некоторую глубину или тень). Это позволит вам увидеть, какие потоки отрисовывают какие фрагменты экрана. На быстрой системе это может быть не так интересно, но если вы усилите отрисовку, растянув изображение до очень большого размера, вы должны увидеть некоторые вариации.

Блокировки

Пакет `java.util.concurrent.locks` содержит классы, которые эмулируют и расширяют встроенные средства синхронизации, добавляя помимо прочего блокировки для «чтения/записи». Как мы упоминали, эти классы являются утилитами, написанными на Java, и напрямую ничего нового не добавляют к семантике языка. Однако они действительно обеспечивают более гибкое использование за счет некоторой встроенной безопасности синхронизации языка Java.

Основу пакета `locks` составляют интерфейсы `Lock` и `Condition`. `Lock` представляет ту же идею, что и блокировка Java (блок контроля), которая связана с каждым объектом и классом для использования с синхронизированными методами и блокировками. Класс `Lock` предоставляет эксклюзивный доступ владельца блокировки, позволяя только одной стороне иметь блокировку в определенный момент времени посредством методов `lock()` и `unlock()`. В синхронизации языка Java это выполняется косвенно при помощи ключевого слова `synchronized`:

```
// синхронизированный метод
synchronized void writeData() { ... }
// синхронизированный блок
synchronized ( someObject ) {
...
}
```

При входе в синхронизированный метод или блок Java приобретает блокировку и автоматически отпускает ее при выходе. Даже если выбрасывается исключение или поток погибает неожиданно, Java автоматически освобождается от всех блокировок, которые приобрела. Использование класса Lock вместо этого требует от нас (или позволяет нам, в зависимости от того, как вы это воспринимаете), явно блокироваться, когда нам нужен ресурс, и вспоминать о разблокировке, когда мы закончили. Блокировка не имеет определенных рамок, как, например, отдельный метод или блок кода. Чтобы воспроизвести эффект синхронизированного метода в примере, мы используем что-то вроде:

```
Lock lock = new ReentrantLock();
// метод или блок
lock.lock();
try {
// тело метода или блока...
```

```
} finally {
lock.unlock()
}
```

Первый оператор вызова `lock()` получает блокировку и продолжает. Последующие вызовы другими потоками блокируются, пока блокировка не будет отпущена. Мы выполняем тело нашей заблокированной операции в блоке `try/finally`. Это, в общем, важно для того, чтобы мы всегда могли разблокироваться перед выходом, но вы можете спокойно реализовать произвольные протоколы на свой страх и риск.

Реализация блокировки в этом примере называется `ReentrantLock`. Имя предполагает, что этот тип блокировки ведет себя как блокировки Java в отношении того, что блокировка связана с потоком оператора вызова. Владелец блокировки может заново получить блокировку («повторно запереть замок») столько раз, сколько он хочет. Например, рекурсивный метод, который блокирует ресурс при входе, — это нормально.

В дополнение к стандартно выглядящему методу `lock()` интерфейс `Lock` имеет методы `tryLock()`, которые не блокируют или блокируют на определенное время, чтобы получить блокировку. Эти условные и ожидающие с ограничением во времени формы являются тем, что обычные замки Java не предоставляют. Реализация интерфейса `ReentrantLock` также имеет понятие «объективности», которое может быть включено или выключено, когда он создается. Когда справедливость восстановлена, блокировка пытается передать блокировку вызывающим методам в том порядке, в котором они ее запрашивают. Обычные замки Java (и дефолтная необъективная политика `ReentrantLock`) не обеспечивают такую гарантию.

Блокировки чтения и записи

Интерфейс `ReadWriteLock` предоставляет доступ к двух различным блокировкам — одна предназначена для записи, а другая для чтения. Идея, лежащая в основе блокировок для чтения/записи, заключается в том, что для многих ресурсов это нормально — когда многие «читатели» просматривают данные, пока они не изменяются. Наоборот, записывающий данные обычно требует эксклюзивного доступа к ним. Именно это делают блокировки для чтения/записи. Любое количество читателей может получить блокировку для чтения, пока имеется блокировка для записи. Только одна сторона может обладать блокировкой записи и ни один читатель не может обладать блокировкой для чтения, пока

блокировка записи отсутствует. Записывающему потоку, возможно, придется подождать, пока не завершится чтение, прежде чем он сможет получить блокировку для записи, а читающий поток, в свою очередь, должен будет подождать, пока не завершится запись:

```
ReadWriteLock rwLock = new ReentrantReadWriteLock( fair );
// поток чтения 1
rwLock.readLock().lock();
// поток чтения 2
rwLock.readLock().lock();
// поток записи
rwLock.writeLock().lock(); // блокировки на потоках 1 и 2
```

В этом отрывке два читателя обладают блокировками чтения, пока пишущий блокирует ожидание замка записи. Когда оба читателя использовали `unlock()` для своих блокировок чтения, писатель получает эксклюзивный доступ к замку и все последующие блокировки чтения, пока пишущий не закончит.

Обладатель замка записи может также приобрести блокировку чтения, но не наоборот. Приобретение блокировки чтения и затем освобождение от замка записи называется *прогибом* замка. Попытки получить блокировку для записи, имея блокировку для чтения (повышение) не разрешены и могут вызвать взаимную блокировку.

Условия

Чтобы завершить описание `Lock` как подобия синхронизации языка Java, нам нужен аналог для механизмов `wait()`, `notify()` и `notifyAll()`. Интерфейс `Condition` предоставляет этот функционал при помощи методов `await()`, `signal()` и `signalAll()`. `Condition` ассоциируется с `Lock` при помощи метода блокировки `newCondition()`. В отличие от обычной блокировки Java, `Lock` может иметь множество объектов интерфейса `Condition`, которые представляют многочисленные наборы ожидания потоков.

Метод `await()` класса `Condition` используется как метод `wait()` объекта Java в блоке `synchronized`:

```
Lock lock= ...
Condition condition= lock.newCondition();
lock.lock();
condition.await(); // блок, ожидающийметода signal()
lock.unlock();
```

```
// тем временем в другом потоке...
lock.lock();
condition.signal();
lock.unlock();
```

Как и метод `wait()`, метод `await()` класса `Condition` может вызываться, только когда поток является владельцем блокировки, связанной с условием, и метод `signal()` может быть вызван только другим потоком, который приобрел блокировку. Однако интересно, что в данном случае эти ограничения являются деталями реализации пакета `java.util.concurrent`; любая другая реализация этих классов может потенциально каким-то образом изменить эти ограничения.

За исключением новых блокировок чтения/записи и некоторых методов ожидания блокировки с ограничением во времени может показаться, что пакет `Locks` мало что привносит в Java. Однако если вы погрузитесь в него глубже, то обнаружите, что это также набор инструментов для создания новых типов примитивов синхронизации и конструкций высокого уровня. Пакет блокировок открывает конкретную реализацию механизма синхронизации Java для того, чтобы его можно было переделывать и расширять. Беглый взгляд на классы реализации позволяет обнаружить изящные методы, такие как `getOwner()`, сообщающий вам, какой поток владеет замком или `getReadLockCount()`, сообщающий вам, сколько читателей работают с вашими данными. Многое возможно с открытой реализацией вроде этой, включающей специализированные пакеты синхронизации, способные, например, автоматически определять мертвые замки или настраивать себя, основываясь на внешней информации. Также могут быть случаи, когда использование API явных блокировок, предоставленных этим пакетом, выполняется лучше, чем синхронизация на уровне языка. Далее мы рассмотрим некоторые инструменты синхронизации высокого уровня.

Средства синхронизации

Пакет `java.util.concurrent` добавляет несколько средств синхронизации высокого уровня, позаимствованных из других языков, включая `CountDownLatch`, `Semaphore`, `CyclicBarrier` и `Exchanger`.

Утилита `CountDownLatch`

`CountDownLatch` является очень простой утилитой синхронизации, которая позволяет любому количеству потоков в блоке ожидать, когда

значение в отчете в обратном направлении достигнет 0, перед тем как они будут «отпущены», чтобы продолжить свою деятельность. `CountDownLatch` инициализируется с отсчетом при конструировании. Впоследствии потоки могут блокироваться, используя метод `await()`, или блокироваться на ограниченный промежуток времени, используя версию `await()` с ожиданием с ограничением во времени. Любой действующий поток может уменьшать счетчик в любое время, независимо от того, заблокированы потоки или нет. Когда счетчик достигает 0, все ожидающие потоки разблокируются и продолжают выполняться. Впоследствии любые вызовы `await()` не блокируются и метод `await()` возвращает `false`, указывая, что отсчет прошел. Отсчет нельзя перезапустить.

```
CountDownLatch latch = new CountDownLatch( 2 ); // отсчет с 2
// поток 1
latch.await(); // блокирует поток 1
// поток 2
latch.countDown(); // отсчет равен 1
latch.countDown(); // отсчет равен 0, поток 1 продолжается
```

Замки отсчета используются в самых разнообразных схемах синхронизации, чтобы координировать ряд потоков на один результат или заставить один поток ждать других, чтобы произвести результаты. Позже мы поговорим о похожей утилите `CyclicBarrier`, которая явно ожидает синхронизации ряда потоков, чтобы координировать действие.

Семафор

Семафоры — это очень старая конструкция синхронизации, которая использовалась во многих других языках. В принципе семафор является пулом *разрешений* — неосязаемых разрешений для выполнения каких-то действий. Семафор инициализируется указанным количеством разрешений. Вызывающие операторы могут использовать методы `acquire()` и `release()`, чтобы брать и возвращать эти разрешения. Вызов метода `acquire()`, когда нет доступных разрешений, является причиной того, что вызывающий оператор блокируется, пока разрешение не будет выпущено. Таким образом, например, семафор может использоваться, чтобы ограничить доступ к какому-то ресурсу к указанному количеству потоков:

```
int concurrentReaders = 5;
boolean fair = true;
Semaphore sem = new Semaphore( concurrentReaders, fair );
Data readData() throws InterruptedException {
```

```
sem.acquire();
// чтение данных…
sem.release();
return data;
}
```

В этом отрывке кода метод `readData()` эффективно ограничивает себя до пяти параллельных потоков чтения в любое указанное время. Дополнительные потоки блокируются в методе `acquire()`, пока разрешение не освободится. В этом смысле семафор отчасти похож на блокировку с множеством обладателей. Однако здесь сходство и заканчивается.

В действительности семафор отличается от блокировки в нескольких отношениях. Во-первых, «пул разрешений» на самом деле является просто цифрой. Метод `acquire()` не возвращает никакого реального значения, и не проводится никакая ассоциация с получателем блокировки. Это означает, что любое фактическое поведение блокировки является строго совместным (по соглашению в приложении). Это также означает, что «разрешения» могут приобретаться и отпускаться различными операторами вызова, невзирая на то, кто действительно «приобретал» их. Это просто увеличение или уменьшение числа. Также, поскольку нет действительной ассоциации с «владельцем», семафоры не являются вновь возвращающимися, как настоящие блокировки. То есть, если поток вызывает метод `acquire()` много раз, он просто уменьшает счетчик много раз. Это поведение могло бы быть полезным в некоторых случаях, например, чтобы считать уровни рекурсии для безопасности API, но не как блокировка, в которой один вызывающий оператор «владеет» несколькими разрешениями. В конце концов, поскольку пул разрешений в действительности является просто числом, вызов методов `acquire()` и `release()` вне синхронизации может увеличить пул разрешений за пределами его отправной точки или сделать его меньше ноля. Он даже может быть инициализирован отрицательным числом, если вы захотите потребовать ослабления прежде, чем кто-то приобретет разрешение.

В дополнение к `acquire()` `Semaphore` имеет метод `tryAcquire()`, который параллелизирует метод `tryLock()` класса `Lock`. Он возвращается немедленно, приобретая разрешение, если оно было доступно, и вместо этого возвращая `false`. Другая форма `tryAcquire()` принимает период с ограничением во времени. Семафоры также обладают понятием «объективности» в порядке запросов приобретения. По умолчанию заказ на запросы не может быть гарантирован, но если при конструировании `Semaphore` установлен флаг «справедливый», `acquire()` раздает разре-

шения в порядке очереди (FIFO). Согласование заключается в том, что упорядочивание может немного влиять на выполнение в зависимости от реализации.

Класс `CyclicBarrier`

`CyclicBarrier` является точкой синхронизации, где указанное количество связанных потоков встречается после завершения своих действий. Когда все потоки прибыли, опционное распределенное защитное действие может выполняться, и тогда все потоки «выпускаются» для продолжения. Класс называется *циклическим*, поскольку он затем может быть использован снова в случае, когда потоки повторяют свою деятельность таким же образом. `CyclicBarrier` является альтернативой использованию метода `join()`, который собирает потоки только после того, как они выполнились и вернулись от их метода `run()`.

Следующий пример, `SiteTimer`, принимает ряд URL в командную строку и рассчитывает, сколько времени займет соединение с каждым, выводя результаты в отсортированном порядке. Он производит соединения в параллели, используя выделенный поток для каждого сайта, и использует `CyclicBarrier` для потоков, чтобы собирать потоки для каждого временного цикла. Затем он выводит согласованные результаты перед тем, чем все начнется заново. Этот пример также иллюстрирует ряд свойств Java, включая обобщения, библиотеки, форматируемую печать, автоупаковывание и внутренний класс. Хотя мы еще не обсуждали библиотеки или сети, их использование в примере довольно простое, и вы можете вернуться к нему после прочтения соответствующих глав этой книги.

```
import java.util.*;
import java.util.concurrent.*;
import java.net.*;
import java.io.IOException;
public class SiteTimer
{
CyclicBarrier barrier;
List<Result> results = new ArrayList<Result>();
private class Result implements Comparable<Result>
{
Long time;
String site;
Result( Long time, String site ) {
this.time = time;
```

```
this.site = site;
}
public int compareTo( Result r ) { return time.compareTo( r.time ); }
}
static long timeConnect( String site ) {
long start = System.currentTimeMillis();
try {
new URL( site ).openConnection().connect();
} catch ( IOException e ) {
return -1;
}
return System.currentTimeMillis() - start;
}
void showResults() {
Collections.sort( results );
for( Result result : results )
System.out.printf( "%-30.30s : %d\n", result.site, result.time );
System.out.println("------------------");
}
public void start( String [] args )
{
Runnable showResultsAction = new Runnable() {
public void run() {
showResults();
results.clear();
} };
barrier = new CyclicBarrier( args.length, showResultsAction );
for ( final String site : args )
new Thread() {
public void run() {
while( true ) {
long time = timeConnect( site );
results.add( new Result( time, site ) );
try {
barrier.await();
} catch ( BrokenBarrierException e ) { return;
} catch ( InterruptedException e ) { return; }
}
}
}.start();
}
```

```
public static void main( String [] args ) throws IOException {
new SiteTimer().start( args );
}
}
```

Метод `start()` создает барьер, указывая количество потоков, которое должно присутствовать прежде, чем группа полностью прибудет, и действие, которое надо выполнить, когда все потоки готовы. Для каждого сайта создается циклический поток, регулирующий время соединения с сайтом и добавляющий результирующий объект к списку перед блокировкой барьеров метода `await()`. Когда все потоки достигают метода `await()`, барьерное действие запускается, выводя результаты. Затем все потоки отпускаются для начала нового цикла.

Если любой из ожидающих потоков прерывается или останавливается (при использовании версии метода `await()` с временным ожиданием), говорится, что барьер «разрушается», и все ожидающие потоки получают исключение `BrokenBarrierException`. В теории барьер может «починиться» при вызове его метода `reset()`, но это сложно, поскольку только один поток из группы может перезапустить барьер должным образом. Метод `reset()`, когда любой другой поток ожидает, вызывает разрушение барьера и ожидающие потоки снова получают исключение, поэтому, вероятно, лучше начать заново в этой точке.

Еще одна деталь: метод `await()` возвращает целое число, указывающее порядок, в котором потоки подошли к барьеру. Это можно использовать, чтобы разделить работу для следующего повторения потоков. Например, если задачи потоков не идентичны, вы можете использовать число, чтобы «выбрать» ведущий поток или разделить потоки на две или больше групп.

Утилита `Phaser`

Java 7 представила новую утилиту параллельности, называющуюся `Phaser`. `Phaser` очень похожа на `CyclicBarrier`, однако она предоставляет немного больше гибкости. `Phaser` названа так отчасти потому, что она приписывает номер каждому циклу своих потоков (номер фазы). Участвующие потоки и случайные свидетели могут читать этот номер, чтобы следить за активностью в барьере. В утилите `CyclicBarrier` количество потоков, которые отслеживаются барьером, фиксировано; новые потоки не могут присоединиться к группе во время ее существования. Этим `CyclicBarrier` отличается от `Phaser`, где количество участников может меняться во время деятельности.

Метод Exchanger

Exchanger является точкой синхронизации пары потоков для обмена элементами данных. Элемент того же типа передается в каждом направлении при использовании метода exchange(). Первый метод прибывает в блоки Exchanger, ожидая своего товарища. Когда прибывает второй метод, каждый из них получает аргумент другого в методе exchange(). Любое количество фактических потоков может использовать Exchanger; они просто располагаются по парам в каком-то порядке, когда прибывают. Exchanger является общим классом, который параметризируется типом объекта для передачи:

```
Exchanger<ByteBuffer> xchange = new Exchanger<ByteBuffer>();
// поток 1
Buffer nextBuf = xchange.exchange( buffer1 ); // блоки
// поток 2
Buffer nextBuf = xchange.exchange( buffer2 );
// смена буферов, оба потока продолжаются…
```

Шаблон Exchanger изначально полезен для повторного использования объектов данных или буферов между потоками, как указано в этом отрывке кода. Скажем, у вас есть поток чтения, заполняющий буферы данными, и поток записи, записывающий содержимое где-то в буферах. Используя Exchanger, считывающий и записывающий поток могут обмениваться парой буферов туда-сюда, не создавая новые потоки. Это может казаться немного загадочным, но это может применяться при использовании усовершенствованных пакетов ввода/вывода NIO, которые мы обсудим в главах 12 и 13.

Мы должны отметить, что Exchanger похож на SynchronousQueue, который мы обсудим в главе 11, когда будем рассматривать API библиотек. Однако Exchanger передает данные в оба направления, между тем, как SynchronousQueue просто передает элементы в одном направлении.

Атомарные операции

Пакет java.util.concurrent.atomic содержит интересный набор классов-оберток для атомарных операций «все или ничего» над определенными простыми типами и значениями ссылок. Атомарная операция — это что-то вроде транзакции, где некоторая последовательность событий либо выполняется, либо терпит неудачу как элемент, и нет никакой воз-

можности увидеть промежуточное состояние. В этом случае транзакции, о которых мы говорим, являются очень простыми операциями, которые либо устанавливают, либо получают значение, иногда в соединении с простой проверкой или математической операцией. Есть атомарные классы-обертки для следующих типов: логические, целые и длинные значения, а также массивы целых чисел и длинных и ссылки на объекты:

```
AtomicBoolean.java
AtomicInteger.java
AtomicIntegerArray.java
AtomicLong.java
AtomicLongArray.java
AtomicReference.java
AtomicReferenceArray.java
```

Класс `AtomicBoolean` (который, кстати, соревнуется с `java.awt.Robot` название лучшего имени класса в Java) служит хорошим примером. На первый взгляд это кажется противоречивым. В конце концов, обычные операции с логическими типами в Java уже атомарны. Предполагается, что нет возможных «промежуточных» состояний для булевых типов, которые могут быть неверно прочтены любым безоблачным многопоточным кодом (как теоретически может быть для длинных и двойных значений). Вместо этого полезность обертки AtomicBoolean заключается в ее комбинированных операциях: `compareAndSet()` и `getAndSet()`:

```
AtomicBoolean bool = new AtomicBoolean( true );
bool.compareAndSet( expectedValue, newValue );
```

Сначала `compareAndSet()` производит сравнение с ожидаемыми значениями (`true` или `false` в случае с булевым типом), и только если значение совпадает, он назначает новое значение. Интересно, что обе эти операции происходят «атомарно», совместно. А значит, нет вероятности, что кто-то изменит значение между тем, как мы проверили его и назначили новое значение. Может показаться, что вероятность и так была не велика, но это важно для гарантии семантики флагов. Например, предположим, что у нас есть главный «закрывающий» переключатель в нашем приложении и поток, который устанавливает, что он хочет произвести чистку на выходе. Используя `compareAndSet()` для начальной проверки, мы можем гарантировать, что только один поток способен установить флаг и выполнить процедуру.

Метод `getAndSet()` просто назначает новое значение и возвращает старое значение таким же безопасным путем. Немного сложнее уви-

деть, как это применяется к булевому типу, так что давайте перейдем к AtomicInteger и AtomicLong. Эти числовые типы имеют дополнительные арифметические комбинированные операции:

```
int getAndIncrement()
int getAndDecrement()
int getAndAdd(int delta)
int incrementAndGet()
int decrementAndGet()
int addAndGet(int delta)
```

getAndIncrement() увеличивает значение на единицу и затем возвращает предыдущее значение. incrementAndGet() делает обратное, возвращая новое значение. Эти операции очень полезны для создания уникальных серийных номеров. Например:

```
AtomicInteger serialNum = new AtomicInteger(0);
public int nextSerialNumber() {
return serialNum.getAndIncrement();
}
```

Мы могли бы выполнить то же самое посредством синхронизации метода, но это проще и иногда намного быстрее.

Ссылки объектного типа могут также быть обернуты для атомарных операций, включая compareAndSet() и getAndSet(). Класс AtomicReference является общим и параметризированным типом ссылки, которую он оборачивает:

```
AtomicReference<Node> ref = new AtomicReference<Node>( node );
ref.compareAndSet( null, newNode );
```

Слабые реализации

У метода compareAndSet()есть странный двойник по имени weak CompareAndSet(), у которого имеется сомнительное отличие, заключающееся в том, что он просто может не сработать при вызове. Однако когда он не срабатывает, он любезно сообщает вам об этом, возвращая false. Какой в этом смысл? Что ж, позволяя неопределенность, Java имеет возможность делать реализацию слабого метода намного быстрее, чем «определенного». Вы можете зациклить и снова запустить слабый метод вместо этого, и он может улучшить выполнение в каких-то архитектурах. Это все происходит потому, что виртуальная машина Java

может быть способна обозначить эти типы атомарных операций вплоть до выполнения на аппаратном уровне, но ограничения подразумевают, что это тяжело гарантировать.

Обновление полей

Для каждого типа в пакете `atomic` содержится набор утилит-оберток, которые помогают обновлять поля. Эти утилиты используют отражение (смотрите главу 7), чтобы выполнить типы атомарных операций, которые мы рассмотрели ранее в «голых» простых типах, уже не обернутых в их атомарные классы-обертки. Корректоры полей работают с переменными в объекте по имени и типу. Загвоздка в том, что атомарность гарантируется только относительно других операторов вызова, которые используют корректоры полей или обычные атомарные обертки классов. По отношению к другим потокам, которые обращаются к переменным в произвольном порядке, не предоставляется никаких гарантий.

Заключение

Java был одним из первых ведущих языков, обеспечивших поддержку потоков на языковом уровне, и теперь является одним из первых языков, стандартизировавших высокоуровневые поточные утилиты, а также API. Мы подошли к завершению нашего разговора о потоках Java, а также в какой-то степени к окончанию первой части книги. В главах с 1 по 9 мы обсуждали язык Java: его синтаксис и «встроенные» свойства. В оставшейся части этой книги мы сфокусируемся в основном на API и библиотеках, которые составляют остальную часть платформы Java. Мы увидим, что истинное очарование Java заключается в соединении этого простого языка с мощными инструментами и стандартами.

Глава 10
РАБОТА С ТЕКСТОМ

Если вы читали эту книгу последовательно, вы прочли все о базовых языковых конструкциях Java, включая объектно-ориентированные аспекты языка и использование потоков. Теперь пришло время переключить передачу и начать говорить о прикладном программном интерфейсе Java (API), библиотеке классов, которая создает стандартные пакеты Java и включает в себя каждую реализацию Java. Базовые пакеты Java являются одной из его наиболее примечательных особенностей. Многие другие объектно-ориентированные языки имеют похожие свойства, но ни в одном из них нет такого большого набора стандартизированных API и инструментов, какой есть в Java. Это и отражение, и причина успеха Java. В табл. 10.1 перечислены некоторые важные пакеты в API и соответствующие главы в этой книге.

Табл. 10.1. Пакеты API Java

Пакет	Содержимое	Глава
`java.lang`	Базовые классы языка	4–9
`java.lang.reflect`	Отражение	7
`java.util.concurrent`	Утилиты потока	9
`java.text` `java.util.regex`	Интернациональные текстовые классы и постоянные выражения	10
`java.util`	Утилиты и классы библиотек	10–12
`java.io` `java.nio`	Ввод и вывод Ввод и вывод	12 12
`java.net`	Организация сети и классы удаленного вызова методов	13–14
`java.rmi`	Классы удаленного вызова методов	13
`javax.servlet`	Веб-приложения	15
`javax.swing` `java.awt`	Swing GUI и графика 2D	16–20

`java.awt.image`
`javax.imageio`
`javax.media`
	Изображения, звук и видео	21

`java.beans`	JavaBeans API	22
`java.applet`	Applet API	23
`javax.xml`	XML API	24

Как вы видите из табл. 10.1, мы привели некоторые классы в `java.lang` в предыдущих разделах, исследуя базовые языковые конструкции. С этой главы мы откроем набор инструментальных средств Java и приступим к изучению остальной части классов API, начиная с утилит, связанных с текстом, поскольку они фундаментальны для всех типов приложений.

API, связанные с текстом

В этой главе мы рассмотрим наиболее специализированные API Java, связанные с текстом, от простых классов для разбора слов и чисел до форматирования текста, интернационализаций и регулярных выражений. Но поскольку многое из того, что мы делаем при помощи компьютера, ориентировано на работу с текстом, классификация API как строго связанной с текстом может быть немного условной. Некоторые пакеты, связанные с текстом, мы рассмотрим в следующей главе, включая API календаря Java, API свойств и предпочтений пользователя и API сбора данных. Но одни из самых важных инструментов на текстовой арене — это инструменты для работы с расширяемым языком разметки, XML. В главе 24 мы рассмотрим в подробностях XML с языками стилей XSL/XSLT. Вместе они предоставляют мощный фреймворк для воспроизведения документов.

Строки

Мы начнем с более близкого рассмотрения Java-класса `String` (или, если быть более точными, `java.lang.String`). Поскольку работа с классами `String` фундаментальна, важно понимать, как они реализуются и что вы можете с ними делать. Объект `String` воплощает последовательность символов Unicode. Изнутри эти символы хранятся в простом массиве Java, но объект `String` ревностно охраняет этот массив и дает к нему доступ только через собственную API. Это необходимо для поддержки идеи о том, что классы `String` являются неизменными; когда вы создаете объект `String`, вы не можете менять его значение. Кажется, что множество операций с объектом `String` меняют символы или длину строки, но в действительности они возвращают новому объекту `String`, который копирует внутренние ссылки, необходимые символы оригинала. Реализации Java делают попытку объединить идентичные строки в одном классе в пул с общей строкой и разделить части классов `String`,

где это возможно. Изначально мотивацией всего этого была производительность. Неизменные классы `String` могут сохранять память и быть оптимизированы под скорость виртуальной машины Java. Обратной стороной медали является то, что программист должен иметь базовое представление о классе `String`, чтобы избежать создания избыточного числа объектов `String` в местах, где производительность является проблемой. Это было особенно справедливо в прошлом, когда виртуальные машины были медленными и слабо контролировали память. Сегодня использование строк обычно не оказывает негативного влияния на эффективность работы реального приложения*.

Создание строк

Текстовые строки, определенные в вашем исходном коде, объявляются двойными кавычками и могут назначаться переменной типа `String`:

```
String quote = "Быть или не быть";
```

Java автоматически конвертирует текстовые строки в объект `String` и назначает его переменной.

Классы `String` следят за своей длиной, поэтому объекты класса `String` не нуждаются в специальных ограничителях. Вы можете получить длину `String` при помощи метода `length()`. Вы также можете сделать проверку на строку с нолевой длиной, используя `isEmpty()`:

```
int length = quote.length();
boolean empty = quote.isEmpty();
```

Строки могут использовать преимущество только из одного перегружаемого оператора в Java, оператора + для объединения строк. Следующий код производит эквивалентные строки:

```
String name = "Джон " + "Смит";
String name = "Джон ".concat("Смит");
```

Текстовые строки не могут изменять строки в исходных файлах Java, но мы можем объединять строки, чтобы создавать такой же эффект:

---

* В случае сомнения проверьте это утверждение! Если ваш код, управляющий String, чистый и легкий для понимания, не перезаписывайте его, пока кто-то не докажет вам, что он медленный. Велика вероятность того, что они неправы. И не обманывайтесь относительными сравнениями. Милисекунда в 1000 раз медленнее, чем микросекунда, но она все еще может быть ничтожной для общей работы вашего приложения.

```
String poem =
" Я помню чудное мгновенье:\n" +
" Передо мной явилась ты,\n" +
" Как мимолетное виденье,\n" +
" Как гений чистой красоты.\n";
```

Вряд ли вы захотите вставлять очень длинный текст в исходный код. В этой и следующей главе мы поговорим о способах загрузки классов `String` из файлов, специальных пакетов, называемых пакетами ресурсов и `URL`. Такие технологии, как серверные страницы Java и шаблонизаторы, также предоставляют способ факторизовать большие объемы текста из вашего кода.

Например, в главе 14 мы увидим, как загрузить нашу поэму с веб-сервера, открывая URL-адрес таким образом:

```
InputStream poem = new URL(
"http://myserver/~dodgson/jabberwocky.txt").openStream();
```

В дополнение к созданию строк из текстовых выражений вы можете создавать класс `String` непосредственно из массива символов:

```
char [] data = new char [] { 'Л', 'е', 'м', 'м', 'и', 'н', 'г' };
String lemming = new String( data );
```

Вы также можете создать класс `String` из массива байтов:

```
byte [] data = new byte [] { (byte)97, (byte)98, (byte)99 };
String abc = new String(data, "ISO8859_1");
```

В этом случае второй аргумент конструктора `String` является именем схемы расшифровки символов. Конструктор классов `String` использует его для преобразования необработанных байтов в указанной кодировке в используемые внутренне двубайтные символы Unicode. Если вы не указываете кодировку символов, используется кодировка по умолчанию для вашей системы. Мы рассмотрим подробнее кодировку символов, когда будем говорить о классе `Charset`, IO, в главе 12*.

Наоборот, метод `charAt()` класса `String` позволяет вам обращаться к символам объекта `String` на манер массива:

```
String s = "Ньютон";
for ( int i = 0; i < s.length(); i++ )
System.out.println( s.charAt( i ) );
```

---

* В операционной системе Mac OS X дефолтной кодировкой является MacRoman. В Windows это CP1252. На некоторых платформах Unix — ISO8859_1.

Этот код выводит символы строки по одному в единицу времени. В ином случае мы можем получить все символы сразу при помощи `toCharArray()`. Вот способ избежать набора кучи одинарных кавычек и получить массив, содержащий алфавит:

```
char [] abcs = "abcdefghijklmnopqrstuvwxyz".toCharArray();
```

Создание строк на основе различных типов

Объекты и простые типы в Java могут быть превращены в стандартные текстовые представления наподобие объекта `String`. Для таких примитивных типов, как номера, строки должны быть достаточно очевидными; для типов объектов это находится под контролем самого объекта. Мы можем получить представление элемента в виде строки при помощи статического метода `String.valueOf()`. Различные перегруженные версии этого метода принимают каждый простой тип:

```
String one = String.valueOf( 1 ); // integer, "1"
String two = String.valueOf( 2.384f ); // float, "2.384"
String notTrue = String.valueOf( false ); // boolean, "false"
```

Все объекты в Java имеют метод `toString()`, который наследуется от класса `Object`. Для многих объектов этот метод возвращает полезный результат, который показывает содержимое объекта. Например, метод `toString()` объекта `java.util.Date` возвращает дату, которую он представляет отформатированной, как строка. Для объектов, не предоставляющих репрезентации, результат строки является просто уникальным идентификатором, который может использоваться для отладки. Метод `String.valueOf()` при вызове для объекта вызывает метод объекта `toString()` и возвращает результат. Одним из реальных отличий в использовании этого метода является то, что если вы передадите его нулевому указателю объекта, он вернет вам «ноль» класса `String` вместо создания исключения `NullPointerException`:

```
Date date = new Date();
// Эквивалентно, например, "Пт Дек 19 05:45:34 1969"
String d1 = String.valueOf( date );
String d2 = date.toString();
date = null;
d1 = String.valueOf( date ); // "null"
d2 = date.toString(); // NullPointerException!
```

Объединение строк использует метод `valueOf()` внутренне, так что если вы «добавите» объект или примитив, используя оператор сложения (+), вы получите класс `String`:

```
String today = "Сегодняшняя дата:" + date;
```

Вы можете увидеть, как некоторые используют пустую строку и оператор сложения (+) в качестве сокращения для получения строкового значения объекта. Например:

```
String two = "" + 2.384f;
String today = "" + new Date();
```

Сравнение строк

Стандартный метод `equals()` может проверять строки на эквивалентность, то есть определять, содержат ли они точно такие же символы в том же порядке. Вы можете использовать другой метод, `equalsIgnoreCase()`, чтобы проверить эквивалентность строк способом, не чувствительным к регистру:

```
String one = "FOO";
String two = "foo";
one.equals( two ); // false
one.equalsIgnoreCase( two ); // true
```

Распространенной ошибкой новичков в программировании на Java является сравнение строк при помощи оператора ==, когда они намереваются использовать метод `equals()`. Вспомните, что строки в Java являются объектами, а == проверяет идентичность объектов; то есть то, являются ли два тестируемых аргумента одним и тем же объектом. В Java легко создать две строки, которые содержат одни и те же символы, но не одни и те же строчные объекты. Например:

```
String foo1 = "foo";
String foo2 = String.valueOf( new char [] { 'f', 'o', 'o' } );
foo1 == foo2 // false!
foo1.equals( foo2 ) // true
```

Эта ошибка особенно опасна, поскольку она часто работает для общего случая, в котором вы сравниваете текстовые строки (строки, объявленные с двойными кавычками прямо в коде). Причина в том, что Java пытается управлять строками эффективно, объединяя их. Во время компиляции Java находит все идентичные строки в заданном классе и создает

только один объект для них. Это безопасно, поскольку строки неизменны и не могут меняться. Вы можете объединять строки самостоятельно этим способом во время выполнения, используя метод `intern()` класса `String`. Интернирование строки возвращает эквивалентную ссылку на строку, которая является уникальной для всей виртуальной машины.

Метод `compareTo()` сравнивает лексическое значение одного класса `String` с другим классом `String`, определяя, сортируется ли он в алфавитном порядке раньше, так же или позже, чем целевая строка. Он возвращает целое число, которое меньше, равно или больше ноля:

```
String abc = "abc";
String def = "def";
String num = "123";
if ( abc.compareTo( def ) < 0 ) // true
if ( abc.compareTo( abc ) == 0 ) // true
if ( abc.compareTo( num ) > 0 ) // true
```

Метод `compareTo()` сравнивает строки строго по их расположению символа в спецификации Unicode. Это работает для простого текста, но не рассматривает все вариации языка верно. Класс `Collator`, который мы обсудим позже, может использоваться для более сложных сравнений.

Класс `Collator`

Пакет `java.text` предоставляет сложный набор классов для сравнения строк в определенных языках. В немецком, например, есть гласные с умляутом и еще один символ, который похож на греческую букву бета и обозначает двойную s. Как нам следует это сортировать? Хотя правила сортировки символов точно определены, вы не можете допустить, что лексическое сравнение, которое мы использовали ранее, имеет правильное значение для других языков, кроме английского. К счастью, класс `Collator` заботится о подобных сложных проблемах сортировки.

В следующем примере мы используем `Collator`, разработанный для сравнения немецких строк. Вы можете приобрести дефолтный `Collator`, вызывая метод `Collator.getInstance()` без аргументов. Когда у вас есть подходящий экземпляр `Collator`, вы можете использовать его метод `compare()`, который возвращает значения прямо как метод `compareTo()` класса `String`. Следующий код создает две строки для перевода на немецкий слов fun и later, используя константы Unicode для этих двух особых символов. Затем он сравнивает их, используя класс `Collator`

для немецкого языка. (Классы `Locale` помогают вам решать проблемы, связанные с отдельными языками и культурами; мы подробно поговорим о них в этой главе.) Результатом в этом случае является то, что fun (Spaß) отсортируется перед later (später):

```
String fun = "Spa\u00df";
String later = "sp\u00e4ter";
Collator german = Collator.getInstance(Locale.GERMAN);
if (german.compare(fun, later) < 0) // true
```

Использование сортировщиков важно, если вы работаете с другими языками, кроме английского. В испанском, например, ll и ch расцениваются как уникальные символы и упорядочиваются по алфавиту отдельно. Сортировщик справляется с такими вещами автоматически.

Поиск

Класс `String` предоставляет несколько простых методов для поиска фиксированных подстрок внутри строки. Методы `startsWith()` и `endsWith()` сравнивают строку аргументов с началом и концом класса `String` соответственно.

```
String url = "http://foo.bar.com/";
if ( url.startsWith("http:") ) // true
```

Метод `indexOf()` ищет первое появление символа или подстроки и возвращает позицию первого символа или –1, если подстрока не найдена:

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 'p' ); // 15
int i = abcs.indexOf( "def" ); // 3
int I = abcs.indexOf( "Fang" ); // -1
```

Подобным образом `lastIndexOf()` ищет в обратном порядке в строке последнее появление символа или подстроки.

Метод `contains()` решает очень распространенную задачу проверки на то, содержится ли данная подстрока в целевой строке:

```
String log = "Возникла опасность в секторе 7!";
if ( log.contains("опасность") ) pageSomeone();
// эквивалентно
if ( log.indexOf("опасность") != -1 ) ...
```

Для более сложного поиска вы можете использовать API регулярных выражений, которая позволяет вам искать и анализировать сложные шаблоны. Мы поговорим о регулярных выражениях позже в этой главе.

Редактирование

Многие методы выполняют операции над `String` и возвращают новый класс `String` в качестве результата. Хоть это и полезно, вы должны знать, что создание большого количества строк таким образом влияет на производительность. Если вам нужно часто изменять строку или создавать сложную строку из компонентов, вы должны использовать класс `StringBuilder`, как мы вскоре увидим.

`trim()` является полезным методом, который устраняет начальный и конечный пробел (например, возврат каретки, перевод строки или позиция табуляции) из `String`:

```
String str = " abc ";
str = str.trim(); // "abc"
```

В этом примере мы выбросили оригинальный класс `String` (с лишними пробелам), и он будет отправлен в мусор.

Методы `toUpperCase()` и `toLowerCase()` возвращают новый класс `String` с нужным регистром:

```
String down = "FOO".toLowerCase(); // "foo"
String up = down.toUpperCase(); // "FOO"
```

`substring()` возвращает указанный диапазон символов. Начальный индекс является включающим; конечный — исключающм:

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
String cde = abcs.substring( 2, 5 ); // "cde"
```

Метод `replace()` предоставляет простую, текстовую замену строки. Одно или больше появлений целевой строки замещаются замещающей строкой, двигающейся от начала до конца. Например:

```
String message = "Привет, NAME, как дела?".replace( "NAME", "Вася" );
// "Привет, Вася, как дела?"
String xy = "xxooxxxoo".replace( "xx", "X" );
// "XooXxoo"
```

Класс `String` также имеет два метода, которые позволяют вам делать более сложную замену по шаблону: `replaceAll()` и `replaceFirst()`. В отличие от простого метода `replace()` эти методы используют регулярные выражения (специальный синтаксис), чтобы описать шаблоны замены, которые мы обсудим позже в этой главе.

Краткое описание методов объекта `String`

Табл. 10.2 кратко излагает методы, предоставленные классом `String`.

Табл. 10.2. Методы класса `String`

Метод	Функционал
`charAt()`	Получает отдельный символ в строке
`compareTo()`	Сравнивает строку с другой строкой
`concat()`	Объединяет строку с другой строкой
`contains()`	Проверяет, включает ли строка другую строку
`copyValueOf()`	Возвращает массив, эквивалентный указанному массиву символов
`endsWith()`	Проверяет, заканчивается ли строка указанным суффиксом
`equals()`	Сравнивает строку с другой строкой
`equalsIgnoreCase()`	Сравнивает строку с другой строкой, игнорируя регистр
`getBytes()`	Копирует символы из строки в байтовый массив
`getChars()`	Копирует символы из строки в массив символов
`hashCode()`	Возвращает хеш-код для строки
`indexOf()`	Ищет первое появление символа или подстроки в строке
`intern()`	Выбирает уникальный экземпляр строки из глобального пула общих строк
`isEmpty()`	Возвращает `true`, если строка нолевой длины
`lastIndexOf()`	Ищет последнее появление символа или подстроки в строке
`length()`	Возвращает длину строки
`matches()`	Определяет, совпадает ли целая строка с шаблоном регулярного выражения
`regionMatches()`	Проверяет, совпадает ли часть строки с указанной частью другой строки
`replace()`	Замещает все появления символа в строке другим символом
`replaceAll()`	Замещает все появления шаблона регулярного выражения шаблоном
`replaceFirst()`	Замещает первое появление шаблона регулярного выражения шаблоном

Метод	Функционал
`split()`	Разделяет строку на массив строк, используя шаблон регулярного выражения в качестве разделителя
`startsWith()`	Проверяет, начинается ли строка с указанного префикса
`substring()`	Возвращает подстроку строки
`toCharArray()`	Возвращает массив символов строки
`toLowerCase()`	Преобразовывает строку в нижний регистр
`toString()`	Возвращает значение строки объекту
`toUpperCase()`	Преобразовывает строку в верхний регистр
`trim()`	Устраняет начальный и конечный пробел строки
`valueOf()`	Возвращает строчное представление значения

Объекты `StringBuilder` и `StringBuffer`

В отличие от строк, предназначенных только для чтения, класс `java.lang.StringBuilder` представляет собой буфер символов, который можно изменять и расширять. Вы можете эффективно использовать его для создания большой строки. `StringBuilder` и `StringBuffer` являются близнецами; у них в точности одинаковый интерфейс API. `StringBuilder` был добавлен в Java 5.0 как входящая, несинхронизированная замена `StringBuffer`. Вскоре мы вернемся к этому.

Сначала давайте посмотрим на некоторые примеры создания объекта `String`:

```
// могло бы быть лучше
String ball = "Эй,";
ball = ball + " привет.";
ball = ball + " Как дела";
```

Этот пример создает ненужный объект `String` каждый раз, когда мы используем оператор объединения (+). Является ли это важным, зависит от того, как часто код запускается и насколько большой фактически становится строка. Вот более экстремальный пример:

```
// Плохое использование + ...
while( (line = readLine()) != EOF )
text += line;
```

Этот пример постоянно производит новые объекты `String`. Массив символов должен копироваться снова и снова, что может отрицательно

влиять на выполнение. Решение заключается в использовании объекта StringBuilder и метода append():

```
StringBuilder sb = new StringBuilder("Эй,");
sb.append(" привет.");
sb.append(" Как дела?");
StringBuilder text = new StringBuilder();
while( (line = readline()) != EOF )
text.append( line );
```

Здесь StringBuilder эффективно справляется с расширением массива при необходимости. Мы можем получить String из StringBuilder при помощи метода toString():

```
String message = sb.toString();
```

Вы также можете вернуть часть StringBuilder как String, используя один из методов substring().

Возможно, вам будет интересно узнать, что, когда вы пишете длинное выражение, используя объединение строк, компилятор генерирует код, который использует StringBuilder за кулисами:

```
String foo = "Быть " + "или " + "не";
```

В действительности это эквивалентно:

```
String foo = new
StringBuilder().append("Быть ").append("или ").append("не").toString();
```

В этом случае компилятор знает, что вы пытаетесь сделать, и заботится об этом за вас.

Класс StringBuilder предоставляет ряд переопределенных методов append() для добавления любого типа данных в буфер. StringBuilder также предоставляет ряд переопределенных методов insert() для вставки различных типов данных в определенное место в буфере строки. Более того, вы можете удалить отдельный символ или ряд символов при помощи методов deleteCharAt() и delete().

В конце концов, вы можете заменить часть StringBuilder содержимым String, используя метод replace(). Классы String и StringBuilder взаимодействуют таким образом, что в некоторых случаях нет необходимости делать никаких копий данных; строчные данные делятся между методами.

Вам следует использовать `StringBuilder` вместо `String` всегда, когда нужно продолжать добавлять символы к строке; он разработан, чтобы обрабатывать такие модификации эффективно. Вы можете преобразовать `StringBuilder` в `String`, когда вам это нужно, или просто объединить и вывести в любом месте, где вы бы использовали `String`.

Как мы говорили раньше, класс `StringBuilder` был добавлен в Java 5.0 в качестве замены `StringBuffer`. Единственное реальное отличие между ними двумя заключается в том, что методы в `StringBuffer` являются синхронизированными, а методы `StringBuilder` — нет. Это означает, что если вы хотите использовать `StringBuilder` из нескольких потоков параллельно, вы должны синхронизировать доступ самостоятельно (что легко выполнимо). Причиной замены является то, что простое использование, как правило, не требует синхронизации и не должно платить никакой связанный с этим штраф (пусть и незначительный).

Интернационализация

Язык Java позволяет нам писать код, который работает одинаково везде, где установлена виртуальная машина JVM. Но на глобальном рынке это только полдела. Большой вопрос остается: будут ли содержимое приложения и данные понятны для конечных пользователей во всем мире? Должны ли пользователи знать английский, чтобы использовать ваше приложение? Java предоставляет всестороннюю поддержку локализации текста вашего приложения для многих современных языков и диалектов. В этой главе мы поговорим о концепциях интернационализации (часто используется аббревиатура I18N) и классах, которые их поддерживают.

Класс `java.util.Locale`

Поддержка интернационализации основана на классе `Locale`. Сам класс очень прост; он включает код страны, код языка и редко используемый вариантный код. Широко используемые языки и страны определены как константы в классе `Locale`. (Забавно, что все эти имена на английском.) Вы можете получить эти коды и читаемые имена как описано ниже:

```
Locale l = Locale.ITALIAN;
System.out.println(l.getCountry()); // IT
System.out.println(l.getDisplayCountry()); // Italy
```

```
System.out.println(l.getLanguage()); // it
System.out.println(l.getDisplayLanguage()); // Italian
```

Коды страны соответствуют ISO 3166. Вы найдете полный список кодов страны в европейском региональном регистраторе Интернета. Языковые коды соответствуют ISO 639. Полный список языковых кодов находится `online` на правительственном сайте США. Нет официального набора вариантных кодов; они разработаны в зависимости от производителя или платформы. Вы можете получить массив поддерживаемых классов `Locale` при помощи статического метода `getAvailableLocales()` (который можете использовать, чтобы позволить вашим пользователям выбирать). Или вы можете получить дефолтный `Locale` для местоположения, где ваш код работает при помощи статического метода `Locale.getDefault()` и позволить вашей системе решать за вас.

Многие классы в Java API используют `Locale`, чтобы решить, как представить текст. Мы столкнулись с таким классом раньше, когда говорили о сортировке текста при помощи класса `Collator`. Далее в этой главе мы увидим больше примеров его применения для форматирования номеров и строк с валютой, а в следующей главе — с классом `DateFormat`, который использует классы `Locale`, чтобы определить, как анализировать даты и время. Вот простой пример без углубления в подробности:

```
System.out.printf( Locale.ITALIAN, "%f\n", 3.14 ); // "3,14"
```

Предыдущее выражение использует итальянский класс `Locale`, чтобы указать, что десятичное число 3.14 должно форматироваться, как оно форматировалось бы на итальянском с использованием запятой вместо десятичной точки. Мы еще поговорим о форматировании текста в этой главе.

Наборы ресурсов

Прежде, чем мы перейдем к деталям или сообщениям форматирования и значениям, мы сделаем шаг назад и зададим серьезный вопрос: что же насчет самих сообщений? Как мы можем писать и управлять приложениями, которые действительно мультиязычны в своих интерфейсах и во всех сообщениях, которые они показывают пользователю? Мы можем найти языковые настройки, но как мы можем управлять всем текстом приложения в нашем коде? Класс `ResourceBundle` предлагает чистое гибкое решение, которое факторизирует текст и ресурсы вашего приложения в классы, ориентированные на конкретный язык или текстовые файлы.

Класс ResourceBundle является коллекцией объектов, к которым ваше приложение может обратиться по имени. Он действует во многом сходно с коллекциями Hashtable или Map, которые мы обсудим в главе 11, исследуя объекты, основанные на классах String, служащих ключами. Класс ResourceBundle данного имени может быть определен для многих разных классов Locale. Чтобы получить определенную ResourceBundle, вызывайте метод генерации ResourceBundle.getBundle(), который принимает имя ResourceBundle и Locale. Следующий пример получает ResourceBundle с именем "Message" для двух классов Locale; из каждого пакета он получает сообщение, чьим ключем является "Hello-Message" и выводит сообщение:

```
import java.util.*;
public class Hello {
public static void main(String[] args) {
ResourceBundle bun;
bun = ResourceBundle.getBundle("Message", Locale.ITALY);
System.out.println(bun.getString("HelloMessage"));
bun = ResourceBundle.getBundle("Message", Locale.US);
System.out.println(bun.getString("HelloMessage"));
}
}
```

Метод getBundle() выбрасывает исключение выполнения MissingResourceException, если подходящий ResourceBundle не может быть определен.

Вы можете предоставить ResourceBundles двумя способами: как компилированные классы Java (жестко закодированный Java) или как простые файлы собственности. Пакеты ресурсов, реализованные как классы, являются либо подклассами ListResourceBundle, либо прямыми реализациями ResourceBundle. Пакеты ресурсов, подтвержденные файлом собственности, представлены во время выполнения объектом PropertyResourceBundle. ResourceBundle.getBundle() возвращает либо подходящий класс, либо экземпляр PropertyResourceBundle, соответствующий подходящему файлу собственности. Алгоритм, используемый getBundle(), основан на добавлении страны и языковых кодов запрашиваемого класса Locale к имени ресурса. Точнее говоря, он ищет ресурсы в таком порядке:

```
name_language_country_variant
name_language_country
```

```
name_language
name
name_default-language_default-country_default-variant
name_default-language_default-country
name_default-language
```

В этом примере, когда мы пытаемся получить `ResourceBundle` с именем `Message`, специально для `Locale.ITALY`, он ищет следующие имена (в классах `Locale`, которые мы используем, нет вариантных кодов):

```
Message_it_IT
Message_it
Message
Message_en_US
Message_en
```

Давайте определим `Message_it_IT ResourceBundle` как жестко запрограммированный класс, подкласс `ListResourceBundle`:

```
import java.util.*;
public class Message_it_IT extends ListResourceBundle {
public Object[][] getContents() {
return contents;
}
static final Object[][] contents = {
{"HelloMessage", "Buon giorno, world!"},
{"OtherMessage", "Ciao."},
};
}
```

`ListResourceBundle` делает легким определение класса `Resource Bundle`; все, что нам нужно сделать, это переопределить метод `getContents()`. Этот метод просто возвращает двумерный массив, содержащий имена и значения его ресурсов. В этом примере `contents[1] [0]` является вторым ключом (`OtherMessage`) и содержимое `[1][1]` является соответствующим сообщением (`Ciao.`).

Давайте определим `ResourceBundle` для `Locale.US`. В этот раз мы пойдем легким путем и создадим файл собственности. Сохраните следующие данные в файле, называемом *Message_en_US.properties:*

```
HelloMessage=Hello, world!
OtherMessage=Bye.
```

Итак, что же происходит, если кто-то запускает вашу программу в `Locale.FRANCE` и `ResourceBundle` не определен для этой местности? Чтобы избежать исключения `MissingResourceException`, стоит определить `ResourceBundle`. В нашем примере вы можете изменять имя файла собственности на *Message.properties*. Таким образом, если `ResourceBundle`, ориентированный на конкретный язык или страну, не может быть найден, ваше приложение все равно может работать (прибегая к своему английскому представлению).

Разбор и форматирование текста

Разбор и форматирование текста является большой и сложной темой. Пока в этой главе мы рассмотрели только простые операции со строками — создание, базовое редактирование, поиск и превращение простых значений в строки. Теперь мы бы хотели перейти к более структурированным формам текста.

В Java есть большой набор инструментов для разбора и вывода отформатированных строк, включая числа, даты, элементы и значения валют. Большинство их мы рассмотрим в этой главе, но с обсуждением форматирования даты и времени подождем до главы 11.

Мы начнем с парсинга — чтения простых чисел и значений в качестве строк и дробления длинных строк на токены. Затем мы зайдем с другой стороны и посмотрим на форматирование строк и пакет `java.text`. Мы вновь пересмотрим тему интернационализации, чтобы увидеть, как Java может локализировать парсинг и форматирование текста, чисел и дат для отдельных областей.

В конце концов, мы детально рассмотрим регулярные выражения, самый мощный инструмент для парсинга текста, который предлагает Java. Регулярные выражения позволяют вам определять ваши собственные шаблоны произвольной сложности, искать их и парсить из текста.

Мы должны упомянуть, что вы увидите много общего между новыми API для форматирования и парсинга (`printf` и `Scanner`), представленными в Java 5.0, и более старыми API пакета `java.text`. Новые API эффективно замещают множество старых и в какой-то степени более легки в использовании. Несмотря на это, хорошо узнать о тех и тех, поскольку в большей части существующего кода используются более старые API.

Разбор простых чисел

В Java числа и логические значения являются простыми типами, а не объектами. Но для каждого простого типа Java также определяет простые классы обертки. Так, пакет `java.lang` включает следующие классы: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` и `Boolean`. Мы говорили о них в главе 1, но возвращаемся к ним снова, потому что эти классы содержат методы статических утилит, которые знают, как парсить их соответствующие типы из строк. Каждый из этих простых классов оберток имеет статический метод «парсинга», который считывает класс `String` и возвращает соответствующий простой тип. Например:

```
byte b = Byte.parseByte("16");
int n = Integer.parseInt( "42" );
long l = Long.parseLong( "99999999999" );
float f = Float.parseFloat( "4.2" );
double d = Double.parseDouble( "99.99999999" );
boolean b = Boolean.parseBoolean("true");
// Prior to Java 5.0 use:
boolean b = new Boolean("true").booleanValue();
```

С другой стороны, `java.util.Scanner` предоставляет одну API не только для парсинга индивидуальных простых типов из строк, но и для чтения их из потоков токенов. Этот пример показывает, как использовать его вместо предшествующих классов оболочек:

```
byte b = new Scanner("16").nextByte();
int n = new Scanner("42").nextInt();
long l = new Scanner("99999999999").nextLong();
float f = new Scanner("4.2").nextFloat();
double d = new Scanner("99.99999999").nextDouble();
boolean b = new Scanner("true").nextBoolean();
```

Мы увидим, что `Scanner` использовался для парсинга многих значений из `String` или потока, когда будем обсуждать размеченный текст позже в этой главе.

Работа с разными системами счисления

Легко разбирать числа целого типа (`byte`, `short`, `int`, `long`) в безопасных числовых базах. Вы можете использовать методы разбора простых классов оберток, просто указав систему счисления как второй параметр:

```
long l = Long.parseLong( "CAFEBABE", 16 ); // l = 3405691582
byte b = Byte.parseByte ( "12", 8 ); // b = 10
```

Все методы в классе `Scanner` в Java 5.0, описанные ранее, также принимают систему счисления в качестве необязательного аргумента:

```
long l = new Scanner( "CAFEBABE" ).nextLong( 16 ); // l = 3405691582
byte b = new Scanner( "12" ).nextByte( 8 ); // b = 10
```

Вы можете пойти другим путем и преобразовать целое значение или значение `long` в строчное значение в указанной системе счисления, используя специальный статический метод `toString()` классов `Integer` и `Long`:

```
String s = Long.toString( 3405691582L, 16 ); // s = "cafebabe"
```

Для удобства каждый класс также имеет статический метод `toHexString()` для работы в шестнадцатеричной системе:

```
String s = Integer.toHexString( 255 ).toUpperCase(); // s = "FF";
```

Числовые форматы

Предшествующие методы оболочки парсера класса обрабатывают случай с числами, отформатированными с использованием только самых простых правил английского языка без украшений. Если эти методы парсинга не понимают строку, поскольку она просто не является действительным числом или поскольку число отформатировано по правилам другого языка, они выбрасывают исключение `NumberFormatException`:

```
// Итальянское форматирование
double d = Double.parseDouble("1.234,56"); // NumberFormatException
```

`Scanner` API умнее и может использовать классы `Locale`, чтобы парсить числа в определенных языках с более совершенными правилами. Например, `Scanner` может обработать числа с запятой:

```
int n = new Scanner("99,999,999").nextInt();
```

Вы можете указать класс `Locale`, отличный от дефолтного, при помощи метода `useLocale()`. Давайте пропарсим это значение в итальянском:

```
double d = new Scanner("1.234,56").useLocale( Locale.ITALIAN
).nextDouble();
```

Если Scanner не сможет парсить строку, он выбрасывает исключение InputMismatchException:

```
double d = new Scanner("garbage").nextDouble(); // InputMismatchException
```

До Java 5.0 этот тип парсинга выполнялся при использовании пакета java.text с классом NumberFormat. Классы пакета java.text также могут позволить вам парсить дополнительные типы, такие как даты, время и локализованные значения валют, которые не обрабатываются при помощи Scanner. Мы рассмотрим это далее в этой главе.

Разметка текста

Распространенная задача в программировании касается парсинга строки текста на слова или «токены», которые разделены некоторым набором символов-разделителей, таких как пробелы или запятые. Первый пример содержит слова, разделенные одними пробелами. Второй, более реалистичный пример, включает поля, разделенные запятыми.

```
Пришло время хороших мужчин (и женщин)...
Check Number, Description, Amount
4231, Java Programming, 1000.00
```

У Java есть несколько (к сожалению, совмещенных) API для обработки таких ситуаций. Наиболее мощными и полезными являются интерфейсы Scanner и метод split() класса String. Оба используют регулярные выражения, чтобы позволить вам разбить строку на произвольные элементы. Мы еще не говорили о регулярных выражениях, но, чтобы показать вам как это работает, мы просто дадим вам необходимую магию и подробно объясним все позже в этой главе. Мы также упомянем унаследованную утилиту, java.util.StringTokenizer, которая использует набор простых символов, чтобы разделить строку. Утилита StringTokenizer не такая мощная, но не требует понимания регулярных выражений.

Метод split() класса String принимает регулярное выражение, которое описывает разделитель и использует его, чтобы разбить строку на массив классов String:

```
String text = "Пришло время хороших мужчин";
String [] words = text.split("\\s");
// words = "Пришло", "время", "хороших", "мужчин", ...
String text = "4231, Java Programming, 1000.00";
```

```
String [] fields = text.split("\\s*,\\s*");
// fields = "4231", "Java Programming", "1000.00"
```

В первом примере мы использовали регулярное выражение \\s, которое совпадает с единственным символом разделителя (пробел, символ табуляции или возврат каретки). Метод split() вернул массив восьми строк. Во втором примере мы использовали более сложное регулярное выражение \\s*,\\s*, которое соответствует запятой, окруженной любым количеством непрерывных пробелов (возможно, нолем). Это уменьшило наш текст до трех симпатичных маленьких полей.

С новым интерфейсом Scanner API мы можем сделать шаг вперед и парсить числа нашего второго примера, когда мы извлечем их:

```
String text = "4231, Java Programming, 1000.00";
Scanner scanner = new Scanner( text ).useDelimiter("\\s*,\\s*");
int checkNumber = scanner.nextInt(); // 4231
String description = scanner.next(); // "Java Programming"
float amount = scanner.nextFloat(); // 1000.00
```

Здесь мы сказали Scanner использовать наше регулярное выражение в качестве разделителя, а затем вызвали его неоднократно, чтобы парсить каждое поле в качестве его соответствующего типа. Scanner удобен, поскольку он может читать не только из класса String, но непосредственно из источников потоков, таких как InputStreams, Files и Channels:

```
Scanner fileScanner = new Scanner( new File("spreadsheet.csv") );
fileScanner.useDelimiter( "\\s*,\\s* );
// ...
```

Еще одна вещь, которую вы можете сделать со Scanner, — это посмотреть вперед с методами hasNext, чтобы увидеть, идет ли следующий элемент:

```
Scanner fileScanner = new Scanner( new File("spreadsheet.csv") );
fileScanner.useDelimiter( "\\s*,\\s* );
// ...
```

Класс StringTokenizer

Даже хотя класс StringTokenizer, который мы упомянули, теперь является наследственным элементом, не помешает знать о нем, поскольку он был в Java с самого начала и используется в большом количестве

кода. StringTokenizer позволяет вам указать ограничитель как набор символов и совпадает с любым количеством комбинаций этих символов как ограничитель между токенами. Следующий отрывок читает слова нашего первого примера:

```
String text = "Пришло время хороших мужчин (и женщин)…";
StringTokenizer st = new StringTokenizer( text );
while ( st.hasMoreTokens() ) {
String word = st.nextToken();
...
}
```

Мы вызываем методы hasMoreTokens() и nextToken(), чтобы пройтись по словам в тексте. По умолчанию класс StringTokenizer использует стандартные пробельные символы — возврат каретки, переход на новую строку и символы табуляции — в качестве ограничителей. Вы также можете указать собственный набор ограничивающих символов в конструкторе StringTokenizer. Любая непрерывная комбинация указанных символов, которая появляется в целевой строке, прерывается между токенами:

```
String text = "4231, Java Programming, 1000.00";
StringTokenizer st = new StringTokenizer( text, "," );
while ( st.hasMoreTokens() ) {
String word = st.nextToken();
// word = "4231", " Java Programming", "1000.00"
}
```

Это не так чисто, как наш пример с регулярным выражением. Здесь мы использовали запятую как ограничитель, поэтому мы получили дополнительный лидирующий пробельный символ в нашем поле описания. Если бы мы добавили пробел в нашу ограничивающую строку, StringTokenizer разбил бы наше описание на два слова, Java и Programming, а это не то, чего вы хотите. Решением здесь будет использование trim(), чтоы убрать начальный и конечный пробел в каждом элементе.

Форматирование Printf-Style

Стандартным свойством, которое Java унаследовал от языка C, является форматирование строки printf-style. Методы форматирования printf-style утилизируют строки особого формата, встроенные в текст,

чтобы указать механизму форматирования, где разместить аргументы и дать подробное определение конверсиям, макету и выравниванию. Методы форматирования printf также пользуются списками аргументов переменной длины, что делает работу с ними намного легче. Вот быстрый пример вывода, отформатированного printf:

```
System.out.printf( "Меня зовут %s и мне %d лет\n", name, age );
```

Форматирование printf получило свое имя от функции printf() языка C, так что если вы программировали на C, это покажется вам знакомым. Java расширил концепцию, добавив некоторую вспомогательную безопасность типов и удобные свойства. Хотя в Java ранее были некоторые возможности форматирования текста (мы обсудим пакет java.text и MessageFormat далее), форматирование printf в действительности не было реализуемым, пока в Java 5.0 не появились списки аргументов переменной длины и автоархивация простых типов. (Мы упомянули это, чтобы объяснить, почему эти одинаковые API обе существуют в Java.)

Программа форматирования

Первичным новым инструментом в нашем арсенале форматирования текста является класс java.util.Formatter и его метод format(). Несколько удобных методов могут скрывать от вас объект Formatter, и вам не нужно непосредственно создавать Formatter. Во-первых, статический метод String.format() может использоваться для форматирования класса String с аргументами (как метод sprintf() в языке C):

```
String message =
String.format("Меня зовут %s и мне %d лет.", name, age );
```

Далее классы java.io.PrintStream и java.io.PrintWriter, которые используются для записи текста в потоки, имеют их собственный метод format(). Мы обсудим потоки в главе 12, но это просто означает, что вы можете использовать то же самое форматирование printf-style для записи строк в любой тип потока, будет ли это стандартный консольный вывод System.out либо файл или сетевое соединение.

Строка форматирования

Синтаксис форматирующей строки является компактным и немного неявным вначале, но вы его оцените, когда к нему привыкнете. Простей-

шая форматная строка представляет собой просто знак (%), за которым следует символ преобразования. Например, следующий текст имеет две встроенные форматирующие строки:

```
"Меня зовут %s и мне %d лет."
```

Первый знак преобразования — `s`, самый общий формат, который представляет значение строки, а второй — `d`, который представляет целое число. Существует около дюжины основных символов преобразования, соответствующих разным типам и примитивам, и на пару дюжин больше символов, которые специально используются для форматирования даты и времени. Сейчас мы разберемся с основами и вернемся к форматированию даты и времени в главе 11.

На первый взгляд может показаться, что некоторые символы преобразования многого не делают. Например, общее преобразование строки `%s` в нашем предыдущем примере могло бы показывать цифровой аргумент возраста, как и `%d`. Однако эти специализированные символы преобразования выполняют три вещи. Во-первых, они добавляют уровень безопасности типов. Устанавливая `%d`, мы обеспечиваем то, что только целый тип форматируется в этом местоположении. Если мы совершим ошибку в аргументах, мы получим исключение `IllegalFormatConversionException` вместо мусора в нашей строке (и ваша IDE также может это отметить). Во-вторых, метод формата чувствителен к классу `Locale` и способен показывать числа, проценты, даты и время на многих разных языках, просто указывая `Locale` в качестве аргумента. Сообщая `Formatter` тип аргумента со специфическими для типа символами преобразования, `printf` может принять во внимание локализацию, ориентированную на конкретный язык. В-третьих, дополнительные флаги и поля могут использоваться для управления макетом с различными значениями для разных типов аргументов. Например, при помощи чисел с плавающей запятой вы можете указать точность в форматирующей строке.

Общий вид встроенной форматирующей строки следующий:

```
%[argument_index$][flags][width][.precision]conversion_type
```

Следующим за символом `%` является число дополнительных элементов перед символом типа преобразования. Мы обсудим их, когда они появятся, но здесь есть процедура завершения. `argument index` может применяться для перестраивания или повторного использования индивидуальных аргументов в списке аргументов переменной длины при

ссылке на них по числу. Поле `flags` содержит один или более специальных символов меток, управляющих форматом. Поля `width` и `precision` управляют размером вывода текста и количеством цифр, показываемых в числах с плавающей запятой.

Преобразования строк

Символ преобразования `s` представляет общий тип преобразования строки. В конце концов, все типы преобразования используют `String`. Мы имеем в виду, что общее преобразование строк идет по простому пути преобразования ее аргумента в строку. Обычно это просто означает вызов `toString()` у объекта. Поскольку все аргументы списка переменного аргумента являются автозапакованными, они все относятся к классу `Object`. Любые примитивы представляются результатами вызова `toString()` на их классы-обертки, что обычно возвращает значение, как вы и ожидаете. Если аргумент нолевой, результатом будет значение класса `String` «`null`».

Более интересными являются объекты, которые реализуют интерфейс `java.util.Formattable`. Для них вызывается метод аргумента `formatTo()`, передающий ему флаги, ширину и точную информацию и позволяющий ему вернуть строку для использования. Таким образом, объекты могут управлять их собственным представлением строки `printf` так же, как это может делать объект, используя `toString()`.

Ширина, точность и выравнивание

Для простых текстовых аргументов вы можете считать ширину и точность минимальным и максимальным количеством символов для вывода. Как мы увидим позже, для числовых типов с плавающей запятой точность медленно изменяет значение и управляет количеством цифр, показанным после десятичной дроби. Мы можем увидеть эффект на простой строке в этом примере:

```
System.out.printf("Строкой является '%5s'\n", "A");
// Строкой является '    A'
System.out.printf("Строкой является '%.5s'\n", "Happy Birthday!");
// Строкой является 'Happy'
```

В первом случае мы указали длину в пять символов, и в результате добавляются пробелы, чтобы перегрузить наш аргумент. Во втором примере мы использовали буквенное значение символа `.`, за которым

следует точное значение из пяти символов, чтобы ограничить длину показанной строки, поэтому наша строка Happy Birthday обрезается после первых пяти символов.

Когда наша строка была перегружена, она была выровнена по правому краю (были добавлены начальные пробелы). Вы можете управлять этим при помощи буквенного символа-флажка минус (-). Изменим наш пример:

```
System.out.printf("Строкой является '%-5s'\n", "A");
// строкой является 'A '
```

И конечно, мы можем указать флаг выравнивания и минимальную и максимальную ширину, объединив их. Вот пример, который выводит слова меняющейся ширины в две колонки:

```
String [] words =
new String [] { "abalone", "ape", "antidisestablishmentarianism" };
System.out.printf( "%-10s %s\n", "Word", "Length" );
for ( String word : words )
System.out.printf( "%-10.10s %s\n", word, word.length() );
// ВЫВОД
Word Length
abalone 7
ape 3
antidisest 28
```

Верхний регистр

«Старший брат» преобразования s (`S`) указывает, что результату преобразования должен быть задан верхний регистр. Несколько символов других простых и числовых преобразований следуют за этим шаблоном, как мы увидим далее. Например:

```
String word = "abalone";
System.out.println(" The lucky word is: %S\n", word );
// The lucky word is: ABALONE
```

Числовые аргументы

Вы можете сослаться на произвольный аргумент по числу из форматной строки, используя обозначение `%n$`. Например, следующий отрывок кода использует один аргумент три раза:

```
System.out.println( "A %1$s это %1$s это %1$S...", "роза" );
// A роза — это роза это роза
```

Числовые аргументы полезны по двум причинам. Первая, показанная здесь, — с их помощью можно использовать аргумент повторно в разных местах и с разными преобразованиями. Полезность этого становится более очевидной, когда мы смотрим на форматирование типа `Date` и `Time` в главе 11, где мы можем ссылаться на один и тот же элемент полдюжины раз, чтобы получить индивидуальные поля. Второе преимущество заключается в том, что числовые аргументы дают сообщению гибкость для переупорядочивания аргументов. Это важно, когда вы используете форматирующие строки, чтобы расположить сообщение с целью интернационализации или кастомизации, где преобразование может диктовать разный порядок.

Простые и числовые преобразования

Табл. 10.3 отображает символ и символы булевых преобразований.

Табл. 10.3. Символ и булевы символы преобразования

Преобразование	Тип	Описание	Пример вывода
`c`	Символьный	Форматирует результат как символ Unicode	`a`
`b`, `B`	Логический	Форматирует результат как логическое значение	`true`, `FALSE`

Символ преобразования `c` производит символ Unicode:

```
System.out.printf("Первая буква %c\n", 'a' );
```

Символы преобразования `b` и `B` выводят логическое значение своих аргументов. Если аргумент равен неопределенному значению, выводом будет `false`. Как ни странно, если аргумент другого типа, отличного от булевого, выводом является `true`. `B` идентично `b`, кроме того, что принуждает вывод в верхнем регистре.

```
System.out.printf( "The door is open: %b\n", ( door.status() == OPEN ) );
```

Что касается типов `String`, значение ширины может быть указано в преобразованиях `c` и `b`, чтобы перегрузить результат до минимальной длины. В табл. 10.4 подводятся итоги символов преобразований целого типа.

Табл. 10.4. Символы преобразования целого типа

Преобразование	Тип	Описание	Пример вывода
`D`	Целый	Форматирует результат как целое число	`999`
`x, X`	Целый	Форматирует результат как шестнадцатеричное число	`FF, 0xCAFE`
`o`	Целое	Форматирует результат как восьмеричное целое	`10, 010`
`h, H`	Целое или объект	Форматирует объект как шестнадцатеричное число. Если объект не является целым, форматирует его значение `hashCode()` или «ноль» для отсутствующего значения	`7a71e498`

Символы преобразования `d`, `x` и `o` содержат значения целого типа `byte`, `short`, `int` и `long`. (`d`, очевидно, представляет десятичное значение, что имеет мало смысла в этом контексте.) Преобразование `h` является особенностью, вероятно, предназначенной для отладки. Несколько важных флагов дают дополнительное управление над форматированием этих числовых типов. Подробности смотрите в разделе «Флаги» далее в этой главе.

Значение ширины может уточняться в этих преобразованиях, чтобы перегрузить результат. Точные значения не разрешены для целых преобразований.

Табл. 10.5 перечисляет символы преобразования типа с плавающей запятой.

Табл. 10.5. Символы преобразования типа с плавающей запятой

Преобразование	Тип	Описание	Пример вывода
`f`	С плавающей запятой	Форматирует результат как десятичное число	`3.14`
`e, E`	С плавающей запятой	Форматирует результат в экспоненциальном представлении	`3.000000e+08`
`g, G`	С плавающей запятой	Форматирует результат в десятичном либо экспоненциальном представлении в зависимости от значения и точности	`3.14, 10.0e-15`
`a, A`	С плавающей запятой	Форматирует результат как шестнадцатеричное число с плавающей запятой со значащей частью числа и показателем степени	`0x1.fep7`

Символ преобразования `f` является первичным символом преобразования с плавающей запятой. Преобразования `e` и `g` позволяют форматирование значений в научной записи. Символ преобразования `a` дополняет возможность в Java назначать значения с плавающей запятой с использованием значащей части и экспонентного значения шестнадцатеричного числа, делая возможным побитовое отображение значения с плавающей запятой без неопределенности.

Как обычно, значение ширины может быть использовано, чтобы перегрузить минимальную длину. Точное значение преобразования, как предполагает его название, контролирует количество цифр, отображаемых после десятичной точки в значениях с плавающими запятыми. Значение округляется при необходимости. Если никакое точное значение не указано, по умолчанию оно округляется до шести цифр:

```
printf("float is %f\n", 1.23456789); // плавающее число 1.234568
printf("float is %.3f\n", 1.23456789); // плавающее число 1.235
printf("float is %.1f\n", 1.23456789); // плавающее число 1.2
printf("float is %.0f\n", 1.23456789); // плавающее число 1
```

Символ преобразования `g` определяет, следует ли использовать десятичную или научную запись. Во-первых значение округляется до указанной точности. Если результат меньше, чем 10^{-4} (меньше, чем .0001) или если больше, чем $10^{precision}$ (10 в степени значения точности), оно показывается в научной записи. В противном случае показывается десятичная запись.

Флаги

Табл. 10.6 вкратце описывает флаги, которые используются в форматирующих строках.

Как указано ранее, флаг может использоваться для форматного вывода выравнивания по левому краю. Остальные флаги влияют на вывод числовых типов, как было описано. Флаг альтернативного варианта # может использоваться для вывода восьмеричных и шестнадцатеричных значений с их стандартными префиксами: `0x` для шестнадцатеричного или 0 для восьмеричного:

```
System.out.printf("%1$X, %1$#X", 0xCAFE, 0xCAFE ); // CAFE, 0xCAFE
System.out.printf("%1$o, %1$#o", 8, 8 ); // 10, 010
```

Табл. 10.6. Флаги, которые используются в форматирующих строках.

Флаг	Типы аргументов	Описание	Пример вывода
-	Нет	Результат выровнен по левому краю (перегруженный пробел справа)	`'foo '`
+	Числовой	Прибавляет префикс со знаком + на положительные результаты	`+1`
' '	Числовой	Прибавляет пробел к положительным результатам (выравнивая их с негативными значениями)	`' 1'`
0	Числовой	Перегружает числа начальными нолями, чтобы согласовать с требованием	`000001`
,	Числовой	Форматирует числа запятыми или другими, отражающими специфику `Locale` групповыми символами	`1,234,567`
(	Числовой	Заключает отрицательные числа в круглые скобки (удобство, используемое для показа лимитов)	`(42.50)`
#	x,X,o	Использует альтернативную форму восьмеричного и шестнадцатеричного вывода	`0xCAFE, 010`

Прочие положения

В табл. 10.7 перечислены остальные элементы форматирования.

Табл. 10.7. Прочие форматирующие элементы

Преобразование	Описание
%	Производит буквенный символ % (Unicode `\u0025`)
n	Производит платформно-зависимый разделитель строк (например, переход на новую строку или возврат каретки, переход на новую строку)

Форматирование при помощи пакета `java.text`

Пакет `java.text` включает, кроме всего прочего, набор классов, разработанных для создания и парсинга строчного представления объектов. В этой главе мы поговорим о трех классах: `NumberFormat`, `ChoiceFormat` и `MessageFormat`. В главе 11 описывается класс `DateFormat`. Как мы ска-

зали ранее, классы пакета java.text совпадают в большой степени с возможностями Scanner и printf-style Formatter. Кроме этих новых свойств, ряд областей в парсинге валют, дат и времени может обрабатываться только пакетом java.text.

Класс NumberFormat может использоваться для форматирования и парсинга валюты, процентов или традиционных чисел. NumberFormat является абстрактным классом, но у него есть несколько полезных методов генерации, которые производят форматеры для различных типов чисел. Например, чтобы форматировать или парсить валютные строки, используйте getCurrencyInstance():

```
double salary = 1234.56;
String here = // $1,234.56
NumberFormat.getCurrencyInstance().format(salary);
String italy = // L 1.234,56
NumberFormat.getCurrencyInstance(Locale.ITALY).format(salary);
```

Первый оператор производит американскую зарплату со знаком доллара, запятой для разделения тысяч и точкой в качестве десятичной точки. Вторая строка представляет ту же строку на итальянском со знаком лиры, точкой для разделения тысяч и запятой в качестве десятичной точки. Запомните, что NumberFormat беспокоится только о формате; он не пытается выполнить преобразование валюты. Мы можем пойти другим путем и парсить форматированное значение, используя метод parse(), как мы увидим в следующем примере.

Подобным образом getPercentInstance() возвращает форматер, который вы можете использовать для создания и парсинга процентов. Если вы не укажете Locale при вызове метода getInstance(), используется дефолтный Locale:

```
double progress = 0.44;
NumberFormat pf = NumberFormat.getPercentInstance();
System.out.println( pf.format(progress) ); // "44%"
try {
System.out.println( pf.parse("77.2%") ); // "0.772"
}
catch (ParseException e) {}
```

И если вы просто хотите создать и парсить традиционные числа, используйте NumberFormat, возвращаемый getInstance() или его эквивалентом, getNumberInstance():

```
NumberFormat guiseppe = NumberFormat.getInstance(Locale.ITALY);
// возвращается к Locale.US
NumberFormat joe = NumberFormat.getInstance();
try {
double theValue = guiseppe.parse("34.663,252").doubleValue();
System.out.println(joe.format(theValue)); // "34,663.252"
}
```

Мы используем переменную guiseppe, чтобы парсить число в итальянском формате (точки разделяют тысячи, запятая является десятичным разделителем). Возвращаемый тип метода parse() является Number, поэтому мы используем метод doubleValue(), чтобы получить значение Number как double. Затем мы используем joe, чтобы отформатировать число корректно для дефолтной области (США).

Вот список методов генерации для текстовых форматеров в пакете java.text. Опять же, мы рассмотрим методы DateFormat в следующей главе.

```
NumberFormat.getCurrencyInstance()
NumberFormat.getCurrencyInstance(Locale inLocale)
NumberFormat.getInstance()
NumberFormat.getInstance(Locale inLocale)
NumberFormat.getNumberInstance()
NumberFormat.getNumberInstance(Locale inLocale)
NumberFormat.getPercentInstance()
NumberFormat.getPercentInstance(Locale inLocale)
DateFormat.getDateInstance()
DateFormat.getDateInstance(int style)
DateFormat.getDateInstance(int style, Locale aLocale)
DateFormat.getDateTimeInstance()
DateFormat.getDateTimeInstance(int dateStyle, int timeStyle)
DateFormat.getDateTimeInstance(int dateStyle, int timeStyle, Locale
aLocale)
DateFormat.getInstance()
DateFormat.getTimeInstance()
DateFormat.getTimeInstance(int style)
DateFormat.getTimeInstance(int style, Locale aLocale)
```

К настоящему времени мы видели, как форматируются числа в качестве текста. Теперь рассмотрим класс ChoiceFormat, который ассоциирует область числовых значений с текстом. ChoiceFormat сконструирован

при помощи указания области числовых значений и строк, которые им отвечают. Один конструктор принимает массив double и массив String, где каждая строка отвечает области от соответствующего числа до (но не включая) следующего числа в массиве:

```
double[] limits = new double [] {0, 20, 40};
String[] labels = new String [] {"молодой", "менее молодой", "старый"};
ChoiceFormat cf = new ChoiceFormat(limits, labels);
System.out.println(cf.format(12)); // "молодой"
System.out.println(cf.format(26)); // "менее молодой"
```

Вы можете указать пределы и метки, используя специальную строку в альтернативном конструкторе ChoiceFormat:

```
ChoiceFormat cf = new ChoiceFormat("0#молодой|20#менее
молодой|40#старый");
System.out.println(cf.format(40)); // старый
System.out.println(cf.format(50)); // старый
```

Пары «ограничение — значение» разделены вертикальными чертами (|); и знак номера (#) отделяет каждый предел от его соответствующего значения.

ChoiceFormat очень полезен для обработки преобразования во множественное число в сообщениях, позволяя вам избежать жутких конструкций, таких как: «у вас открыт один файл (файлы)». Вы можете создавать читаемые сообщения об ошибках, используя класс ChoiceFormat вместе с MessageFormat.

Средство MessageFormat

MessageFormat — это средство форматирования, которое использует шаблонную строку так же, как это делает форматирование printf(). MessageFormat в большей степени был замещен printf(), который имеет больше опций и более широко используется вне Java. Тем не менее некоторые до сих пор предпочитают стиль MessageFormat, который чуть менее загадочен, чем стиль printf(). MessageFormat обладает статическим методом форматирования MessageFormat.format(), который распараллеливает форматирование стиля печати String.format().

Аргументы в форматирующей строке MessageFormat очерчены фигурными скобками и могут включать информацию о том, как они долж-

ны быть отформатированы. Каждый аргумент состоит из числа, дополнительного типа и дополнительного стиля, как кратко сформулировано в табл. 10.8.

Табл. 10.8. Аргументы `MessageFormat`

Тип	Стили
`Choice`	*`pattern`*
`Date`	`short`, `medium`, `long`, `full`, *`pattern`*
`Number`	`integer`, `percent`, `currency`, *`pattern`*
`Time`	`short`, `medium`, `long`, `full`, *`pattern`*

Давайте используем пример, чтобы это пояснить.

Специальный указатель `{0}` означает «используйте элемент ноль из аргументов, поставляемых методу `format()`». Когда мы генерируем сообщение, вызывая `format()`, мы передаем значения для замены символов заполнителей (`{0}`, `{1}`, ...) в шаблоне. В этом случае мы передаем строку no в качестве аргумента `arguments[0]`, извлекая результат You have no messages.

Давайте попробуем этот пример снова, но в этот раз мы сформируем число и дату вместо аргумента строки:

```
MessageFormat mf = new MessageFormat(
"У вас {0, number, integer} сообщений на {1, date, long}.");
// "У вас 93 сообщения на Апрель 10, 2002."
System.out.println( mf.format( 93, new Date() ) );
```

В этом примере нам нужно заполнить два пробела в шаблоне, поэтому нам нужно два аргумента. Первый должен быть числом и форматироваться как целое. Второй должен быть `Date` и выводиться в формате `long`.

Это все еще сыро. Что если там только одно сообщение? Чтобы сделать это грамматически правильным, мы можем внедрить строку-шаблон стиля `ChoiceFormat` в нашей строке-шаблоне `MessageFormat`:

```
MessageFormat mf = new MessageFormat(
"У вас {0, number, integer} сообщение{0, choice, 0#s|1#|2#s}.");
// "У вас 1 сообщение."
System.out.println( mf.format( 1 ) );
```

В этом случае мы можем использовать первый аргумент дважды: один раз — чтобы обеспечить количество сообщений и один раз — чтобы обеспечить ввод в шаблон `ChoiceFormat`. Шаблон говорит о добавлении `s`, если у аргумента есть значение 0 или он равен 2 или больше.

При написании интернационализированных программ вы можете использовать пакеты ресурсов, чтобы обеспечивать не только текст сообщений, то и форматирующие строки для ваших объектов `MessageFormat`. Таким образом, вы можете автоматически форматировать сообщения на целевых языках — с датами и другими зависящими от языка полями, которые будут обрабатываться надлежащим образом. Поскольку аргументы в форматирующей строке пронумерованы, вы можете обратиться к ним из любого места. Например, вы можете сказать: «На диске C 123 файла»; на каком-то другом языке вы можете сказать: «123 файла на диске C». Вы можете реализовать оба сообщения с одним и тем же набором аргументов:

```
MessageFormat m1 = new MessageFormat(
"На диске {0} {1, number, integer} файлов.");
MessageFormat m2 = new MessageFormat(
"{1, number, integer} файлов на диске {0}.");
```

В реальной жизни код может быть более компактным; вы бы использовали только один объект `MessageFormat`, инициализированный строкой, взятой из пакета ресурсов. Или вы скорее предпочли бы использовать статический метод `format` или полностью переключиться на `printf()`.

Регулярные выражения

Теперь пришло время сделать небольшой крюк в нашем путешествии по Java и ступить на землю регулярных выражений. Регулярное выражение, или коротко «регулярка», описывает шаблон текста. Регулярные выражения используются со многими инструментами — включая пакет `java.util.regex`, текстовые редакторы и многие скриптовые языки — чтобы обеспечить возможности по сложному поиску текста и мощному управлению строками.

Если вы уже знакомы с понятием регулярных выражений и с тем, как они используются в других языках, вы, возможно, захотите ограничиться беглым просмотром этой главы. Но вам обязательно следует озна-

комиться с разделом API java.util.regex, который рассматривает классы Java, необходимые для их использования. С другой стороны, если в своем путешествии по Java вы добрались до этого места, не имея никаких знаний по данной теме, и интересуетесь именно тем, что же такое регулярные выражения, тогда откупорьте бутылочку вашего любимого напитка и готовьтесь. Скоро вы изучите самый мощный инструмент в арсенале обработки текста, в действительности являющийся языком внутри языка, и все это всего на нескольких страницах.

Язык регулярных выражений

Регулярное выражение описывает шаблон в тексте. Под шаблоном мы понимаем почти каждое свойство, которое вы можете выявить в тексте, начиная с одного буквенного символа (или отдельных буквенных символов), фактически не понимая его смысла. Это включает такие элементы, как слова, словосочетания, строки и параграфы, пунктуация, падеж и, в более широком смысле, строки и числа со специальной конструкцией для них, такие как номера телефонов, адреса электронной почты и цитируемые выражения. При помощи регулярных выражений вы можете искать в словаре все слова, в которых есть буква q без ее подружки u рядом с ней или слова, которые начинаются и заканчиваются одной и той же буквой. Когда вы создали шаблон, вы можете просто использовать инструменты, чтобы искать его в тексте или определять, совпадает ли данная строка с ним. Регулярное выражение также может быть организовано таким образом, чтобы помочь вам разбить на части отдельные фрагменты текста, с которыми оно совпадет и которые вы можете потом использовать как элементы замещаемого текста, если захотите.

Достаточно написать один шаблон

Перед тем как продолжить, мы должны сказать несколько слов о синтаксисе регулярных выражений в общем. В начале этого раздела мы мимоходом упомянули, что будем обсуждать новый язык. Регулярные выражения действительно составляют простую форму языка программирования. Если вы на мгновение задумаетесь о примерах, которые мы приводили ранее, вы увидите, что будет необходимо что-то наподобие языка, чтобы описать даже простые шаблоны — такие как электронные адреса, которые имеют некоторые вариации форм. Учебник по вычислительной технике определил бы регулярные выражения в самый низ

иерархии компьютерных языков — как по тому, что они описывают, так и по тому, что вы можете с ними сделать. Однако они все равно могут быть довольно сложными. Как и во многих языках программирования, элементы регулярных выражений являются простыми, но они могут создавать комбинации произвольной сложности. И вот тут все начинает становиться трудным.

Поскольку регулярные выражения работают со строками, удобно иметь очень компактное обозначение, которое легко можно вбить между символами. Но компактное обозначение может быть очень неявным, и опыт показывает, что намного легче написать сложное выражение, чем перечитать его позже. Таково проклятие регулярных выражений. Бывает, что в момент вдохновения, навеянного поздней ночью и кофеином, вам удается написать один чудесный шаблон, который упростит остальную часть вашей программы до одной строки. Однако когда вы вернетесь, чтобы прочесть эту строку на следующий день, она покажется вам набором египетских иероглифов. То, что проще, обычно лучше. Если вы можете разбить вашу проблему на несколько шагов и сделать ее более понятной, возможно, вам следует так и поступить.

Экранированные символы

Теперь, когда вы надлежащим образом предупреждены, нам придется сообщить вам еще одну вещь перед тем, как мы переместим вас в обратном направлении. Запись регулярного выражения может не только становиться чрезмерно сложной, оно также выглядит немного двусмысленно среди обычных строк Java. Важной частью выражения является экранированный символ, символ с обратным слешем перед ним. Например, экранированный символ `d`, `\d` (`d` с обратным слешем) является сокращением, которое совпадает с любым единичным цифровым символом (0–9). Однако вы не можете просто написать `\d` в качестве строки Java, поскольку Java использует обратный слеш для собственных специальных символов и чтобы уточнить последовательности символов (`\uxxxx`). К счастью, Java предлагает нам замену: экранированный обратный слеш, который представляет собой два обратных слеша (`\\`), означает буквенный обратный слеш. Правило следующее: когда вы хотите, чтобы обратный слеш появился в вашем регулярном выражении, вы должны экранировать его дополнительным слешем:

```
"\\d" // строка Java, которая выдает "d" с обратным слешем
```

И чтобы окончательно все запутать: поскольку в самом регулярном выражении используется обратный слеш для обозначения специальных символов, должен быть такой же «спасательный люк», позволяющий вам удваивать обратные слеши, если вам нужен буквальный обратный слеш. Итак, если вы хотите указать регулярное выражение, которое включает единичный буквальный обратный слеш, он выглядит таким образом:

```
"\\\\" // строка Java выдает два обратных слеша; регулярное выражение
выдает один
```

Большинство «магических» символов оператора, о которых вы прочтете в этом разделе, работают с символом, который предшествует им, поэтому их тоже нужно экранировать, если вы хотите получить их буквальное значение. В том числе символы `.`, `*`, `+`, фигурные скобки `{}`, и круглые скобки `()`.

Если вам нужно создать часть выражения, в котором имеется множество буквальных символов, вам помогут специальные разделители `\Q` и `\E`. Любое появление текста между `\Q` и `\E` автоматически экранируется. (Вам все еще нужны переходы `String` — двойные слеши для обратного слеша, но не четверные.) Также существует статический метод `Pattern.quote()`, который делает то же самое, возвращая верно экранированную версию любой строки, которую вы ему предоставляете. Все, что я могу предложить кроме этого, чтобы помочь вам сохранить рассудок при работе с этими примерами, — это хранить две копии: строку комментария, показывающую чистое регулярное выражение и реальную строку Java, где вы должны удваивать все двойные слеши.

Символы и классы символов

Теперь давайте погрузимся в фактический синтаксис регулярных выражений. Простейшей формой регулярного выражения является простой текст, который не имеет никакого специального значения и прямо совпадает (символ за символом) во вводе. Это может быть один символ или более. Например, в следующей строке шаблон «о» может совпадать с символом «о» в словах «роза» и «стоит»:

```
"Роза стоит $1.99."
```

Шаблон «роза» может соответствовать только буквенному слову `rose`. Но это не очень интересно. Давайте перейдем на ступеньку выше, введя некоторые специальные символы в понятие «классов» символов.

Любой символ: точка(.)

Специальный символ точка (.) совпадает с любым единичным символом. Шаблон «.оза» совпадает с роза, коза, _оза (пробел, за которым идет оза) или любым другим символом, за которым следует последовательность оза. Две точки совпадают с любыми двумя символами и так далее. Оператор . не является дискриминантным; обычно он прекращается только для символа конца строки (и в некоторых случаях вы можете сказать ему, чтобы он не прекращался; мы обсудим это позже).

Мы можем рассмотреть . для представления группы или класса всех символов. И регулярное выражение также определяет более интересные классы символов.

Символ с пробелом или без пробела: \s, \S

Специальный символ \s совпадает с буквенным символом пробела или одним из следующих символов: \t (табуляция), \r (возврат каретки), \n (переход на новую строку), \f (прогон страницы) и обратный ход. Специальный символ \S делает обратное, совпадая с любым символом кроме символов пробела.

Числовой или нечисловой символ: \d, \D

\d совпадает с любым числом от 0 до 9. \D делает обратное, совпадая со всеми символами, кроме чисел.

Словесный или несловесный символ: \w, \W

\w совпадает со «словесным» символом, включая буквы в верхнем и нижнем регистре A–Z, a–z, числа 0–9 и символ подчеркивания (_).\W совпадает со всем, кроме этих символов.

Специализированные классы символов

Вы можете определить собственные классы символов, используя запись [...]. Например, следующий класс совпадает с любыми символами a, b, c, x, y, or z:

```
[abcxyz]
```

Специальная запись диапазона x-y может использоваться как сокращение для символов алфавита. Следующий пример определяет класс символа, содержащий все буквы в верхнем и нижнем регистре:

```
[A-Za-z]
```

Помещение каретки (^) в качестве первого символа внутри скобок инвертирует класс символа. Этот пример совпадает с любым символом, кроме A-F в верхнем регистре:

```
[^A-F] // G, H, I, ..., a, b, c, ... etc.
```

Вложенность классов символов просто добавляет их:

```
[A-F[G-Z]] // A-Z
```

Логическое понятие AND — && — может использоваться для сложения (символов совокупно):

```
[a-p&&[l-z]] // l, m, n, o, p
[A-Z&&[^P]] // от A до Z кроме P
```

Метки расположения

Шаблон «[Aa] rose» (включая A в нижнем и верхнем регистре) совпадает три раза в следующей фразе:

```
"A rose is a rose is a rose"
```

Символы позиции позволяют вам назначать относительное положение совпадения. Самыми важными являются ^ и $, которые совпадают с началом и концом строки, соответственно:

```
^[Aa] rose // совпадает с "A rose" в начале строки
[Aa] rose$ // совпадает "a rose" в конце строки
```

По умолчанию ^ и $ совпадают с началом и концом «ввода», который часто является строкой. Если вы работаете с большим количеством строк текста и вам нужны совпадения с началом и концом строк внутри одной большой строки, мы можете обратиться к «многострочному» режиму, как описано дальше в этой главе.

Маркеры позиции \b и \B совпадают с границей слова или границей несловесного символа соответственно. Например, следующий шаблон совпадает с rose и rosemary, но не primrose:

```
\brose
```

Повторение (множественность)

Простое совпадение с фиксированными символами не слишком много нам даст. Далее мы рассмотрим операторы, которые считают количество появлений символа (или, в более общем смысле, шаблона):

Любое (ноль или больше повторений): звездочка (*). Звездочка (*) после символа или класса символа означает «позволить любое количество этого типа символа» — другими словами, ноль или более. Например, следующий шаблон совпадает с числом с любым количеством предшествующих нолей (возможно, ни одного):

```
0*\d // совпадает с числом с любым количеством предшествующих нолей
```

Несколько (одно или более повторений): знак сложения (+)

Знак сложения (+) означает «одно или более» повторений и эквивалентно XX* (шаблон, за которым следует шаблон со звездочкой). Например, следующий шаблон совпадает с числом с одной или более цифр плюс опциональными предшествующими нолями:

```
0*\d+ // совпадает с числом (с одним или более цифр) с опциональными
предшествующими
// нолями
```

Может показаться излишним выбор нолей в начале выражения, поскольку ноль является цифрой и таким образом выбирается частью выражения \d+ в любом случае. Однако мы покажем позже, как вы можете расщепить строку, используя регулярное выражение и получить как раз те фрагменты, которые вы хотите. В этом случае вы, возможно, захотите удалить предшествующие ноли и сохранить только числа.

Опциональный (нолевое или одно повторение): вопросительный знак (?)

Оператор с вопросительным знаком (?) позволяет ровно ноль или одно повторение. Например, следующий шаблон выбирает дату истечения срока кредитной карточки, в которой может присутствовать или отсутствовать слеш посредине:

```
\d\d/?\d\d // выбирает четыре цифры с опциональным слешем в середине
```

Диапазон (повторения между x и y, включительно): {x,y}

Оператор диапазона в фигурных скобках {x,y} является наиболее распространенным оператором повторений. Он указывает точный диапазон для выборки. Диапазон принимает два аргумента: нижнюю и верхнюю границу, разделенные запятой. Это регулярное выражение выбирает любое слово с пятью или семью символами, включительно:

```
\b\w{5,7}\b // выбирает слова с минимум 5 и максимум 7 символами
```

По крайней мере x или более повторений (y бесконечен): `{x,}`

Если вы опустите верхнюю границу, просто оставив свободно болтающуюся запятую в диапазоне, верхняя страница становится бесконечной. Этот способ позволяет указывать минимум появлений без максимума.

Группирование

Как в логических и математических операциях, в регулярных выражениях могут использоваться круглые скобки, чтобы создать подвыражения или наложить границы на части выражений. Эта сила позволяет нам расширять операции, о которых мы говорили, чтобы работать не только с символами, но и со словами или другими регулярными выражениями. Например:

```
(yada)+
```

Здесь мы применяем оператор + (один или больше) ко всему шаблону yada, не только один символ. Это совпадает с yada, yadayada, yadayadayada и так далее.

Используя группирование, мы можем начать создавать более сложные выражения. Например, хотя многие электронные адреса имеют структуру из трех частей (например, foo@bar.com), на самом деле доменное имя может содержать произвольное количество разделенных точкой компонентов. Чтобы обрабатывать это правильно, мы можем использовать выражение наподобие этого:

```
\w+@\w+(\.\w)+ // Выбрать электронный адрес
```

Это выражение выбирает слово, за которым следует символ @, за которым следует другое слово, а затем одно или более слов, разделенных точкой — например, pat@pat.net, `friend@foo.bar.com` или `mate@foo.bar.co.uk`.

Захват групп

В дополнение к базовому группированию операций круглые скобки играют важную дополнительную роль: текст, выбранный каждым подвыражением, в круглых скобках может возвращаться по отдельности. То есть вы можете изолировать текст, который совпал с каждым подвыражением. Существует специальный символ для каждой захваченной группы в регулярном выражении по номеру. Это важное свойство можно использовать двумя способами.

Во-первых, вы можете создать регулярное выражение, которое обращается к только что выбранному им тексту и использует этот текст как параметр для следующих выборок. Это позволяет вам создавать очень значительные вещи. Например, мы можем показать пример со словарем, который мы упоминали во введении. Давайте найдем все слова, которые начинаются и заканчиваются на одну и ту же букву:

```
\b(\w)\w*\1\b // выбирает слова, начинающиеся и заканчивающиеся одной
и той же буквой
```

Видите 1 в примере? Это ссылка на первую захваченную группу в этом выражении, (\w). Ссылки на группы захвата принимают форму \n, где n – число в захваченной группе, считающее слева направо. В этом примере первая захваченная группа выбирает символ слова на границе слов. Затем мы разрешаем любое количество символов слова до специальной ссылки \1 (за которой также следует граница слова). \1 означает «значение, выбранное в захваченной группе один». Поскольку эти символы должны быть одинаковыми, это регулярное выражение выбирает слова, которые начинаются и заканчиваются одним и тем же символом. Второй вариант использования захваченных групп заключается в ссылке на выбранные части текста при конструировании текста замещения. Мы покажем вам, как это делать, немного позже, когда будем говорить об API регулярных выражений.

Захваченные группы, конечно, могут содержать более одного символа, и количество групп может быть любым. Вы даже можете вкладывать захваченные группы. Дальше мы обсудим в подробностях, как они нумеруются.

Нумерация

Захваченные группы нумеруются от 1 и двигаются слева направо, считая количество открытых круглых скобок, которое следует преодолеть, чтоб добраться до них. Специальный номер группы 0 всегда ссылается на целую выборку выражения. Например, рассмотрите следующую строку:

```
один ((два) (три (четыре)))
```

Эта строка создает следующие выборки:

```
Group 0: один два три четыре
Group 1: два три четыре
Group 2: два
```

```
Group 3: три четыре
Group 4: четыре
```

Перед тем как продолжить, мы должны отметить еще одно. До сих пор в этом разделе мы показывали с лучшей стороны тот факт, что круглые скобки делают двойную работу: создают логические группировки для операций и определяют захваченные группы. Что если две эти роли конфликтуют? Допустим, у нас есть сложное регулярное выражение, которое использует круглые скобки, чтобы группировать подвыражения и создавать захваченные группы? В этом случае вы можете использовать специальный незахватывающий групповой оператор (`?:`), чтобы произвести логическое группирование вместо применения круглых скобок. Возможно, вам не нужно будет делать это часто, но хорошо это знать.

Переключение

Оператор вертикальная черта (`|`) обозначает логическую операцию `OR`, которая также называется переключателем или выбором. Оператор `|` не воздействует на отдельные символы, но вместо этого применяется ко всему с обеих его сторон. Он делит выражение на два, если не ограничивается группировками с круглыми скобками. Например, слегка наивным подходом к парсингу дат может быть следующий:

```
\w+, \w+ \d+ \d+|\d\d/\d\d/\d\d // шаблон 1 или шаблон 2
```

В этом выражении левая часть выбирает такие шаблоны, как Fri, Oct 12, 2001, а правая выбирает 10/12/2001.

Следующее регулярное выражение может использоваться для выбора адресов электронной почты с одним из трех доменных имен *(net, edu и gov)*:

```
\w+@[\w\.]*\.(net|edu|gov) // адрес электронной почты, заканчивающийся на
net, .edu, или .gov
```

Специальные опции

Существует несколько специальных опций, которые влияют на то, как движок регулярного выражения выполняет свою выборку. Эти опции могут быть применены двумя способами:

- вы можете передать один или более флажков во время шага `Pattern.compile()` (будет обсуждаться позже в этой главе);
- вы можете включить специальный блок кода в ваше регулярное выражение.

Мы покажем последний подход. Чтобы это сделать, включите один или более флажков в специальный блок `(?x)`, где *x* является флагом опции, которую мы можем включить. Обычно вы это делаете в начале регулярного выражения. Вы также можете отключить флаги, добавив знак минус `(?-x)`, который позволяет вам применять флаги, чтобы выбирать части вашего шаблона.

Доступны следующие флаги:

Нечувствительные к регистру: `(?i)`

Флаг (`?i`) сообщает движку регулярного выражения о том, что нужно игнорировать регистр при выборке, например:

```
(?i)yahoo // выбирает Yahoo, yahoo, yah00 и др.
```

Точка для всего: `(?s)`

Флаг `(?s)` включает режим «точка для всего», позволяющий символу точки выбирать все, включая символы окончания строки. Это полезно, если вы выбираете шаблоны, которые охватывают несколько строк. `s` означает «однострочный режим», немного сбивающее с толку название, позаимствованное из Perl.

Многострочный: `(?m)`

По умолчанию `^` и `$` в действительности не выбирают начало и конец строки (как определено возвратом каретки или комбинациями перевода на новую строку); вместо этого они выбирают начало или конец всего вводимого текста. Включение многострочного режима при помощи `(?m)` заставляет их выбирать начало и конец каждой строки, а также начало и конец ввода. Конкретно это означает место перед первым символом, место после последнего символа и места сразу после или непосредственно перед ограничителями строки внутри строки.

Строчки Unix: `(?d)`

Флаг `(?d)` ограничивает определение ограничителя строки для специальных символов `^`, `$` и `.` только до перехода на новую строку в стиле Unix. По умолчанию возврат каретки при переходе на новую строку `(\r\n)` тоже разрешен.

Жадность

Мы видели намеки на то, что регулярные выражения способны сортировать некоторые сложные шаблоны. Но есть случаи, когда то, что должно быть выбрано, является неоднозначным (по крайней мере, для

нас, но не для движка регулярных выражений). Вероятно, самый сложный пример имеет дело с количеством символов, которые операторы повторений потребляют перед остановкой. Операция `.*` лучше всего иллюстрирует это. Рассмотрите следующую строку:

```
"Пришло время <bold>действовать</bold>, а не говорить."
```

Предположим, что мы хотим отыскать все теги стиля HTML (часть между символами < и >), например, потому что мы хотим удалить их.

Мы можем исходно начать со следующего регулярного выражения:

```
</?.*> // выбирает <, выборочно /, а затем все до >
```

Затем мы получаем следующую выборку, которая является слишком длинной:

```
<bold>действовать</bold>
```

Проблема заключается в том, что операция `.*`, как все операции повторения, по умолчанию является «жадной», это означает что она потребляет абсолютно все, что может до последнего совпадения оконечного знака (в этом случае >) в файле или строке.

Существуют решения этой проблемы. Первое заключается в том, чтобы «сказать, что это», то есть быть точными в отношении того, что разрешено между скобками. Содержимое HTML-тега на самом деле не может включить всего; например, оно не может включить закрывающую скобку (>). Поэтому мы перезапишем наше выражение как:

```
</?\w*> // выбирает <, выборочно /, любое количество символов слова, then >
```

Но предположим, что содержимое не так легко описать. Например, мы можем искать строки с цитатами в тексте, что может включать практически любой текст. В этом случае мы можем использовать второй подход и «сказать, чем это не является». Мы можем перевернуть нашу логику из предыдущего примера и указать, что все, *кроме* закрывающей скобки, позволено внутри скобок:

```
</?[^>]*>
```

Вероятно, это самый эффективный способ сообщить движку регулярного выражения, что делать. Тогда он точно знает, что искать, чтобы закончить чтение. Однако этот подход имеет ограничения. Не очевидно, как им пользоваться, если ограничитель более сложен, чем единичный символ. Это также не очень изящно.

В конце концов, мы приходим к основному решению: использование противодействующих операторов. Для каждого оператора повторения

существует альтернативная, нежадная форма, которая потребляет настолько мало символов, насколько это возможно, при этом все еще пытаясь получить выборку из того, что идет после. Это именно то, что нам было нужно в нашем предыдущем примере.

Противодействующие операторы принимают форму стандартного оператора с добавленным ?. (Да, мы знаем, что это сбивает с толку.) Теперь мы можем записать наше регулярное выражение таким образом:

```
</?.*?> // выбирает <, выборочно /, минимальное количество символов, затем >
```

Мы добавили ? к .*, чтобы заставить .* выбрать как можно меньше символов, при этом делая финальную выборку >. Та же технология (добавление ?) работает со всеми операторами повторений, как в следующих двух примерах:

```
.+? // один или более, нежадно
.{x,y}? // между x и y, нежадно
```

Просмотр вперед и назад

Чтобы понять следующую тему, давайте ненадолго вернемся к символам, помечающим позицию (^, $, \b и \B), которые мы обсуждали ранее. Подумайте о том, что именно эти пометки делают для нас. Например, мы говорим, что маркер \b выбирает границу слова. Но слово «выборка» может быть здесь чересчур сильным. В действительности он «требует» того, чтобы граница слова появилась в указанном месте регулярного выражения. Допустим, если бы у нас не было \b, как мы могли создать ее? Что ж, мы могли попробовать создать регулярное выражение, которое выбирает границу слова. Это может казаться легким, если есть классы со словесными и несловесными символами (\w и \W):

```
\w\W|\W\w // выбрать начало или конец слова
```

Но что теперь? Мы могли бы попробовать вставить этот шаблон в наши регулярные выражения везде, где мы бы использовали \b, но на самом деле это не одно и то же. В действительности мы выбираем эти символы, а не просто требуем их. Это регулярное выражение выбирает два символа, создающих границу слова в дополнение к любым другим выборкам, следующим после, если операция \b просто требует границы слова, но не выбирает никакого текста. Различие заключается в том, что \b не является шаблоном выборки, но является подобием просмотра вперед. Просмотр вперед — это шаблон, который требуется для выборки

следующего элемента в строке, но не поглощается движком регулярного выражения. Когда шаблон просмотра вперед достигает цели, шаблон движется дальше, и символы остаются в потоке для следующего использования шаблона. Если просмотр вперед не удается, выборка не удается (или она возвращается и пробует другой подход).

Мы можем создавать собственные просмотры вперед при помощи операции просмотра вперед (`?=`). Например, чтобы выбрать букву X в конце слова, мы можем использовать:

```
(?=\w\W)X // Найти X в конце слова
```

Здесь движок регулярного выражения требует шаблон `\W\w`, чтобы выбрать, но не поглотить символы, оставляя их для следующей части шаблона. Это эффективно позволяет нам писать перекрывающие шаблоны (как в предыдущем примере). Например, мы можем выбрать слово Pat, только если оно является частью слова Patrick таким образом:

```
(?=Patrick)Pat // Найти Pat только в Patrick
```

Другая операция (`?!`), негативный просмотр вперед, требует того, чтобы шаблон не делал выборку. Мы можем найти все появления Pat вне Patrick при помощи этого:

```
(?!Patrick)Pat // Найти Pat не в Patrick
```

Не имеет значения, что мы могли бы написать все эти примеры другим способом, просто выбирая большее количество текста. Например, в первом примере мы могли бы выбрать все слово Patrick. Но это не так точно и если бы мы хотели использовать захваченные группы, чтобы вывести выбранный текст или его части позже, нам бы пришлось исхитриться, чтобы получить то, что мы хотим. Например, предположим, что мы хотим заменить что-то для Pat (скажем, поменять шрифт). Нам придется использовать дополнительную захваченную группу и заменить ею текст. Использовать просмотр вперед легче.

В дополнение к просмотру вперед в потоке мы можем использовать операции *просмотра назад*. Например, мы можем найти мою фамилию, но только для случаев, когда речь обо мне:

```
(?<=Pat )Niemeyer // Нимейер, только когда перед ним следует Pat
```

Или мы можем найти строку bean только если она не является частью фразы Java bean:

```
(?<!Java *)bean // Перед словом bean нет слова Java
```

В этих случаях просмотр назад и выбранный текст не пересекались, потому что просмотр назад был перед выбранным текстом. Но вы можете поместить просмотр вперед или просмотр назад в любое место — перед выборкой или после нее — например, мы также могли выбрать Pat Niemeyer таким образом:

```
Niemeyer(?<=Pat Niemeyer)
```

java.util.regex API

Теперь, когда мы изучили в теории то, как создавать регулярные выражения, самое сложное позади. Все, что осталось, это исследовать Java API для применения регулярных выражений: поиска их в строке, получения захваченного текста и замены выборок замещающим текстом.

Шаблон

Как мы сказали, шаблоны регулярных выражений, которые мы записали здесь как строки, в действительности являются маленькими программами, которые описывают, как выбирать текст. Во время выполнения пакет регулярных выражений Java компилирует эти маленькие программы в форму, которую он может выполнить для некоего целевого текста. Несколько простых удобных методов принимают строки непосредственно, чтобы использовать как шаблоны. Однако в более общем смысле Java позволяет вам явно компилировать ваш шаблон и воплощать его в экземпляре объекта класса `Pattern`. Это самый эффективный способ управления шаблонами, которые используются больше одного раза, поскольку он избавляет нас от ненужной повторной компиляции строки. Чтобы скомпилировать шаблон, мы используем статический метод `Pattern.compile()`:

```
Pattern urlPattern = Pattern.compile("\\w+://[\\w/]*");
```

Когда у вас есть класс `Pattern`, вы можете попросить его создать объект `Matcher`, который связывает шаблон с целевой строкой:

```
Matcher matcher = urlPattern.matcher( myText );
```

Сопоставитель выполняет выборки. Мы поговорим об этом далее. Но перед этим упомянем один удобный метод класса `Pattern`. Статический метод `Pattern.matches()` просто берет две строки — регулярное выражение и целевую строку — и определяет, совпадает ли целевая строка

с регулярным выражением. Это очень удобно, если вы хотите осуществить в вашем приложении быструю проверку один раз. Например:

```
Boolean match = Pattern.matches( "\\d+\\.\\d+f?", myText );
```

Эта строка кода может определить, содержит ли строка `myText` число с плавающей запятой в стиле Java, такое как 42.0f. Отметьте, что строка должна полностью совпадать, чтобы считаться совпадением.

Matcher

`Matcher` связывает шаблон со строкой и предоставляет инструменты для проверки, находя и повторяя совпадения шаблона со строкой. `Matcher` является «хранящим состояние». Например, метод `find()` пытается найти совпадение в тексте каждый раз, когда вызывается. Но вы можете очистить `Matcher` и начать сначала, вызвав его метод `reset()`.

Если вы просто заинтересованы в «одном большом совпадении» — то есть вы ожидаете, что ваша строка либо совпадает с шаблоном, либо нет — вы можете использовать `matches()` или `lookingAt()`. Они в общих чертах соответствуют методам `equals()` и `startsWith()` класса `String`. Метод `matches()` спрашивает, совпадает ли строка с шаблоном целиком (не оставляя ни одного символа в строке) и возвращает `true` или `false`. `lookingAt()`. Метод `lookingAt()` делает то же самое, но спрашивает только, начинается ли строка с шаблона, и не заботится о том, использованы ли в шаблоне все символы строки. Как правило, вам будет нужно осуществлять поиск в строке и находить одно и более совпадений. Чтобы это делать, вы можете использовать метод `find()`. Каждый вызов `find()` возвращает `true` или `false` для следующего совпадения в шаблоне и внутренне отмечает позицию совпадающего текста. Вы можете получить позиции символов, которыми начинается и заканчивается совпадение, методами `start()` и `end()` объекта `Matcher` или вы можете просто получить текст при помощи метода `group()`. Например:

```
import java.util.regex.*;
String text="A horse is a horse, of course of course...";
String pattern="horse|course";
Matcher matcher = Pattern.compile( pattern ).matcher( text );
while ( matcher.find() )
System.out.println(
"Matched: '"+matcher.group()+"' at position "+matcher.start() );
```

Предыдущий отрывок выводит начинающую позицию слов horse и course (всего четыре):

```
Matched: 'horse' at position 2
Matched: 'horse' at position 13
Matched: 'course' at position 23
Matched: 'course' at position 33
```

Метод для получения совпавшего текста называется `group()`, потому что он ссылается на захваченную группу ноль (полное совпадение). Вы также можете получить текст других пронумерованных групп захвата, давая методу `group()` целый аргумент. Вы можете определить, как много групп захвата у вас есть, при помощи метода `groupCount()`:

```
for (int i=1; i < matcher.groupCount(); i++)
System.out.println( matcher.group(i) );
```

Разбивка и создание токенов из строки

Очень распространенной потребностью является парсинг строки на группу полей, основанных на каком-то разделителе, таком как запятая. Это такая распространенная проблема, что в Java 1.4 был добавлен метод к классу `String` как раз для этой цели. Метод `split()` принимает регулярное выражение и возвращает массив подстрок, разбитых вокруг этого шаблона. Например:

```
String text = "Foo, bar , blah";
```

`String [] fields = text.split( "\s*,\s*" );` выдает массив `String`, содержащий `Foo`, `bar` и `blah`. Вы можете управлять максимальным количеством совпадений и тем, получаете ли вы «пустые» строки (для текста, который мог возникнуть между двумя смежными ограничителями), используя поле опционального ограничения.

Если вы собираетесь использовать операцию, подобную этой, больше нескольких раз в вашем коде, вам, пожалуй, следует компилировать шаблон и использовать его метод `split()`, который идентичен версии в `String`. Метод `split()` в `String` эквивалентен:

```
Pattern.compile(pattern).split(string);
```

Еще один взгляд на Scanner

Как мы упомянули, вводя класс `Scanner`, в Java 5.0 он может использовать регулярные выражения, чтобы разбивать строки на токены.

Вы можете указать регулярное выражение, чтобы использовать в качестве ограничителя (вместо пробельного символа по умолчанию) либо во время создания, либо применяя метод `Delimiter()`. Методы `next()`, `hasNext()`, `skip()` и `findInLine()` класса `Scanner` все также принимают регулярные выражения. Вы можете определять их как строки или при помощи компилированного объекта `Pattern`.

Вы можете использовать метод `findInLine()` класса `Scanner` как улучшенный `Matcher`. Например:

```
Scanner scanner = new Scanner( "Quantity: 42 items, Price $2.34" );
scanner.findInLine("[Qq]uantity[:\\s]*");
int quantity=scanner.nextInt();
scanner.findInLine("[Pp]rice.*\\$");
float price=scanner.nextFloat();
```

Предыдущий отрывок размещает значения количества и цены, позволяя вариации в применении заглавных букв и пробелах перед числами.

Прежде чем продолжить, мы упомянем «Глупый фокус со сканером», который, хоть мы его и не рекомендуем, вы можете найти забавным. Используя пограничный маркер \A, который обозначает начало ввода, в качестве ограничителя, мы можем сказать классу `Scanner`, чтобы он возвращал целый ввод как одну строку. Это легкий способ читать содержимое любого потока в одной большой строке:

```
InputStream source = new URL("http://www.oreilly.com/").openStream();
String text = new Scanner( source ).useDelimiter("\\A").next();
```

Вероятно, это не самый эффективный и понятный способ, но он может позволить вам набирать поменьше текста в вашем эксперименте.

Замена текста

Распространенной причиной, по которой вы можете начать искать шаблон в строке, является необходимость заменить его на что-то другое. Пакет регулярных выражений не только облегчает этот процесс, но также предоставляет простое определение, чтобы помочь вам создать текст замены, используя куски совпадающего текста.

Самая удобная форма этой API — это метод `replaceAll()` класса `Matcher`, который подставляет строку замены для каждого появления шаблона и возвращает результат. Например:

```
String text = "Richard Nixon's social security number is: 567-68-0515.";
Matcher matcher =
Pattern.compile("\\d\\d\\d-\\d\\d\-\\d\\d\\d\\d").matcher( text );
String output = matcher.replaceAll("XXX-XX-XXXX");
```

Этот код замещает все появления номеров Управления социальным обеспечением Соединенных Штатов символами “XXX-XX-XXXX”(возможно, из соображений приватности).

Использование захваченного текста в замене. Буквальная замена является хорошим способом, но мы можем сделать его более мощным, используя группы захвата в нашем шаблоне замены. Чтобы это сделать, мы используем простое правило обращения к нумерованным группам захвата при помощи понятия `$n`, где *n* является номером группы. Например, предположим, что мы хотим показать какую-то малую часть номера социальной безопасности из предыдущего примера, чтобы пользователь мог узнать, что речь идет о нем. Мы можем изменить наше регулярное выражение, чтобы оно захватило, например, последние четыре цифры таким образом:

```
\d\d\d-\d\d-(\d\d\d\d)
```

Затем мы можем использовать это в тексте замены:

```
String output = matcher.replaceAll("XXX-XX-$1");
```

Статический метод `Matcher.quoteReplacement()` может использоваться, чтобы затемнить буквенную строку (так что он игнорирует обозначение $) перед использованием его в качестве замещающего текста.

Управление заменой. Метод `replaceAll()` является полезным, но вы можете захотеть более точно управлять каждой заменой. Например, заменить каждое совпадение чем-то другим или основывать замену на совпадении каким-то программным способом.

Чтобы сделать это, вы можете использовать методы `appendReplacement()` и `appendTail()`. Эти методы могут использоваться в соединении с методом `find()` во время итераций по совпадениям для создания строки замещения. `appendReplacement()` и `appendTail()` работают с `StringBuffer`, который вы поставляете. Метод `appendReplacement()` создает строку замещения, отслеживая, где вы находитесь в тексте, и добавляя несовместимый текст в буфер, а также заменяя текст, который вы поставляете. Каждый вызов метода `find()` добавляет промежуточный текст из прошлого вызова, за которым следует замещение, затем

прогоняет все совпавшие символы, чтобы подготовиться к следующему вызову. В конце концов, когда вы достигли последнего совпадения, вы должны вызвать метод `appendTail()`, который добавляет любой оставшийся текст после последнего совпадения. Мы покажем пример этого далее, когда будет создавать простой «шаблонизатор».

Наш простой шаблонизатор

Давайте свяжем все, что мы обсудили, в изящном примере. Распространенной проблемой в приложениях Java является работа с громоздким многострочным текстом. Как правило, вы не хотите хранить текст сообщений в коде вашего приложения, потому что их тогда тяжело редактировать и интернационализировать. Но если вы переносите их во внешние файлы или ресурсы, вам необходим способ подключения информации во время выполнения. Лучший пример этого существует в сервлетах Java; созданная страница HTML часто состоит из статического текста на 99% только с несколькими подключенными «переменными» кусками. Такие технологии, как JSP и XSL, разработаны для решения этой проблемы. Но это большие инструменты, а у нас простая проблема. Так что давайте создадим простое решение — шаблонизатор.

Наш шаблонизатор читает тексты, содержащие специальные теги шаблона и замещает значения, которые мы предоставили. И поскольку создание HTML или XML является одним из важнейших областей его применения, мы будем дружелюбными с этими форматами, делая так, чтобы наши теги приспосабливались к стилю комментария XML. А именно, наш движок ищет в тексте теги, что выглядит таким образом:

```
<!--TEMPLATE:name This is the template for the user name -->
```

Комментарии в стиле XML начинаются с символов `<!--` и могут содержать что угодно до закрывающего тега `-->`. Мы создадим правило о необходимости поля `TEMPLATE:name` для указания имени значения, которого мы хотим использовать. Кроме этого, мы позволим любой текст описания, который пользователь захочет добавить. Чтобы быть дружелюбными (и корректными), мы позволим, чтобы в тегах появлялось любое количество пробелов, включая многострочный текст в комментариях. Мы также будем игнорировать регистр текста идентификатора TEMPLATE на всякий случай. Теперь мы можем делать это все с командами `String` низкого уровня, перескакивая пробелы и захватывая множество подстрок. Но используя мощность регулярных выражений,

мы можем сделать все намного чище и только с приблизительно семью строками релевантного кода. (Мы сократили пример с немного большим количеством кода, чтобы сделать его более полезным.)

```
import java.util.*;
import java.util.regex.*;
public class Template
{
Properties values = new Properties();
Pattern templateComment =
Pattern.compile("(?si)<!--\\s*TEMPLATE:(\\w+).*?-->");
public void set( String name, String value ) {
values.setProperty( name, value );
}
public String fillIn( String text ) {
Matcher matcher = templateComment.matcher( text );
StringBuffer buffer = new StringBuffer();
while( matcher.find() ) {
String name = matcher.group(1);
String value = values.getProperty( name );
matcher.appendReplacement( buffer, value );
}
matcher.appendTail( buffer );
return buffer.toString();
}
}
```

Вы можете использовать класс `Template` таким образом:

```
String input = "<!-- TEMPLATE:name --> lives at "
+"<!-- TEMPLATE:address -->";
Template template = new Template();
template.set("name", "Bob");
template.set("address", "1234 Main St.");
String output = template.fillIn( input );
```

В этом коде `input` является строкой, содержащей теги для имени и адреса. Метод `set()` предоставляет значения для этих тегов.

Давайте начнем с расщепления регулярного выражения `template Pattern` в примере:

```
(?si)<!--\s*TEMPLATE:(\w+).*?-->
```

Выглядит страшно, но в действительности это очень просто. Просто начните читать слева направо. Во-первых, у нас есть объявление специальных флагов (`?si`), говорящее движку регулярного выражения, что оно должно быть в режиме одной строки, где `.*` выбирает все символы, включая новые строки (`s`) и игнорируя выбор (`i`). Далее следует буквальное <!—, за которым следует количество пробелов (`\s`) и идентификатор `TEMPLATE:`. После двоеточия мы имеем захваченную группу (`\w+`), которая читает наш идентификатор имени и сохраняет для нас, чтобы вернуть позже. Мы позволяем все (`.*`) до -->, будучи осторожными, указывая, что `.*` должен быть нежадным (`.*?`). Мы не хотим, чтобы `.*` поглощал другие открывающие и закрывающие теги комментариев до последнего, но вместо этого находил самое маленькое совпадение (один тег).

Наш метод `fillIn()` делает работу, принимая строку шаблона, выполняя ее поиск и «замещая» значения тега значениями из `set()`, которое мы сохранили в таблице `Properties`. Каждый раз, когда вызывается `fillIn()`, он создает объект `Matcher`, чтобы обернуть строку ввода и подготовиться к тому, чтобы применить шаблон-образец. Затем он создает временный `StringBuffer`, чтобы хранить вывод и циклы, используя метод `find()` объекта `Matcher`, чтобы получить каждый тег. Для каждого совпадения он возвращает значение захваченной группы (группа один), которая хранит имя тега. Он ищет соответствующее значение и замещает тег этим значением в буфере строки вывода, используя метод `appendReplacement()`. (Вспомните, что `appendReplacement()` заполняет промежуточный текст при каждом вызове, поэтому нам не нужно это делать.) Все, что остается, это вызвать `appendTail()` в конце, чтобы получить оставшийся текст после последнего совпадения и вернуть значение строки. Вот и все!

Мы надеемся, что этот раздел показал вам некоторые возможности, которые дают эти инструменты, и подогрел ваш аппетит для продолжения. Регулярные выражения позволяют вам работать так, как вы раньше не могли себе представить. Особенно сейчас, когда мир программного обеспечения сфокусировался на текстовом представлении практически всего — от данных до интерфейсов пользователя — посредством XML и HTML, в которых инструменты по управлению текстом являются фундаментальными. Просто помните, что нужно оставлять эти регулярного выражения простыми, чтобы можно было использовать их снова и снова.

Глава 11
ОСНОВНЫЕ ИНСТРУМЕНТЫ

В этой главе мы продолжим рассматривать API-интерфейсы языка Java, охватывая все больше инструментов из пакета `java.util`. Этот пакет содержит широкий спектр вспомогательных средств, включая инструменты для математических операций и фундаментальные структуры данных (коллекции); он поддерживает работу с датой и временем, хранение пользовательских настроек и ведение журнала сообщений.

Математические инструменты

Java поддерживает арифметические операции с целыми и действительными числами прямо на уровне языка. Более высокоуровневые математические операции реализуются при помощи класса `java.lang.Math`. Как вы уже могли заметить, классы-обертки для простых типов данных позволяют обращаться с этими типами как с объектами. Классы-обертки также обладают некоторыми методами для выполнения простейших преобразований.

Прежде всего, стоит сказать несколько слов об арифметических операциях, встроенных в язык Java. Ошибки, возникающие на уровне целочисленной арифметики, Java обрабатывает путем генерирования исключения `ArithmeticException`:

```
int zero = 0;
try {
   int i = 72 / zero;
} catch ( ArithmeticException e ) {
   // деление на ноль
}
```

Чтобы сгенерировать ошибку в этом примере, мы создали вспомогательную переменную `zero`. Компилятор обладает определенной проницательностью; если бы мы явно попытались поделить на числовую константу, равную нолю, он бы поймал нас на горячем.

С другой стороны, арифметические операции над числами с плавающей запятой не генерируют исключений. Вместо этого они принимают специальные значения, свидетельствующие о выходе за пределы допустимого диапазона, как показано в табл. 11.1.

Табл. 1.11. Специальные значения для чисел с плавающей запятой

Значение	Математическое представление
`POSITIVE_INFINITY`	`1,0/0,0`
`NEGATIVE_INFINITY`	`-1,0/0,0`
`NaN`	`0,0/0,0`

Следующий пример выдает бесконечный результат:

```
double zero = 0.0;
double d = 1.0/zero;
if ( d == Double.POSITIVE_INFINITY )
   System.out.println( "Деление на ноль" );
```

Специальное значение `NaN` (сокращенно от not a number — не число) является результатом деления ноля на ноль и обладает уникальной математической особенностью: оно не равно самому себе (выражение `NaN != NaN` возвращает `true`).

Чтобы выполнить проверку на это значение, используйте метод `Float.isNaN()` или `Double.isNaN()`.

Класс `java.lang.Math`

Класс `java.lang.Math` — это математическая библиотека языка Java. Он содержит набор статических методов, охватывающих все стандартные математические операции, такие как `sin()`, `cos()` и `sqrt()`. Класс `Math` не совсем объектно-ориентированный (вы не можете создать его экземпляр). На самом деле это просто удобное хранилище статических методов, которые больше похожи на глобальные функции.

Как мы уже видели в главе 6, благодаря возможностям статического импорта подобные методы и константы можно импортировать прямо в область видимости своего класса и использовать их, указывая простые неполные имена.

В табл. 11.2 подытожены методы из класса `java.lang.Math`.

Табл. 11.2. Методы класса `java.lang.Math`

Метод	Типы аргументов	Выполняемые функции
`Math.abs(a)`	`int, long, float, double`	Абсолютное значение
`Math.acos(a)`	`double`	Арккосинус
`Math.asin(a)`	`double`	Арксинус
`Math.atan(a)`	`double`	Арктангенс
`Math.atan2(a,b)`	`double`	Угловая часть преобразования декартовых координат в полярные
`Math.ceil(a)`	`double`	Наименьшее целое число, которое больше или равно `a`
`Math.cbrt(a)`	`double`	Кубический корень из `a`
`Math.cos(a)`	`double`	Косинус
`Math.cosh(a)`	`double`	Гиперболический косинус
`Math.exp(a)`	`double`	Значение `Math.E` в степени `a`
`Math.floor(a)`	`double`	Наибольшее целое число, которое меньше или равно `a`
`Math.hypot(a,b)`	`double`	Точное вычисление квадратного корня из `a2 + b2`
`Math.log(a)`	`double`	Натуральный логарифм `a`
`Math.log10(a)`	`double`	Десятичный логарифм `a`
`Math.max(a, b)`	`int, long, float, double`	Значение `a` или `b`, которое ближе всего к константе `Long.MAX_VALUE`
`Math.min(a, b)`	`int, long, float, double`	Значение `a` или `b`, которое ближе всего к константе `Long.MIN_VALUE`
`Math.pow(a, b)`	`double`	`a` в степени `b`
`Math.random()`	`Отсутствуют`	Генератор случайных чисел
`Math.rint(a)`	`double`	Приводит значение типа `double` к ближайшему целому в формате `double`
`Math.round(a)`	`float, double`	Округляет до целого числа
`Math.signum(a)`	`double, float`	Преобразует числа с разными знаками в `1,0, -1,0` или `0`
`Math.sin(a)`	`double`	Синус
`Math.sinh(a)`	`double`	Гиперболический синус
`Math.sqrt(a)`	`double`	Квадратный корень
`Math.tan(a)`	`double`	Тангенс
`Math.tanh(a)`	`double`	Гиперболический тангенс
`Math.toDegrees(a)`	`double`	Переводит радианы в градусы
`Math.toRadians(a)`	`double`	Переводит градусы в радианы

Методы `log()`, `pow()` и `sqrt()` во время выполнения могут генерировать исключение `ArithmeticException`. Методы `abs()`, `max()` и `min()` имеют перегруженные версии для всех скалярных значений (`int`, `long`, `float` или `double`), возвращая соответствующие типы. Версии метода `Math.round()` принимают значения типа `float` или `double`, возвращая, соответственно, `int` или `long`. Остальные методы принимают и возвращают значения типа `double`.

```
double irrational = Math.sqrt( 2.0 ); // 1.414...
int bigger = Math.max( 3, 4 ); // 4
long one = Math.round( 1.125798 ); // 1
```

Для удобства класс `Math` также содержит статические финальные значения типа `double`, `E` и `PI`:

```
double circumference = diameter * Math.PI;
```

Большие/точные числа

На случай если типы `long` и `double` для вас недостаточно большие или точные, пакет `java.math` предоставляет два полнофункциональных класса, `BigInteger` и `BigDecimal`, которые поддерживают числа произвольной точности. Они имеют целый ряд методов для выполнения математических операций произвольной точности и для строгого контроля над округлением остатка. В следующем примере мы используем класс `BigDecimal` для сложения двух больших чисел и выполнения деления, результат которого будет иметь 100 знаков после запятой:

```
long l1 = 92233720368547758O7L; // Long.MAX_VALUE
long l2 = 92233720368547758O7L;
System.out.println( l1 + l2 ); // -2 ! Не годится.
try {
   BigDecimal bd1 = new BigDecimal( "92233720368547758O7" );
   BigDecimal bd2 = new BigDecimal( 92233720368547758O7L );
   System.out.println( bd1.add( bd2 ) ); // 184467440737095516l4
   BigDecimal numerator = new BigDecimal(1);
   BigDecimal denominator = new BigDecimal(3);
   BigDecimal fraction =
         numerator.divide( denominator, 100, BigDecimal.ROUND_UP );
   // дробь с сотней знаков после запятой = 0.333333 ... 3334
}
catch (NumberFormatException nfe) { }
catch (ArithmeticException ae) { }
```

Если вы когда-нибудь ради забавы будете реализовывать криптографические или научные алгоритмы, класс `BigInteger` станет для вас незаменимым.

Компоненты числа с плавающей запятой

Как мы уже упоминали в главе 4, для внутреннего представления чисел с плавающей запятой (типы `float` и `double`) Java использует стандарт IEEE 754. Те из вас, кто знаком с принципом работы арифметики с плавающей запятой, уже знают, что данный стандарт предусматривает двоичное представление «десятичных» чисел путем разделения их на три части: знак (положительный или отрицательный), экспоненциальное представление *модульной величины* в степени кратной 2, а также *мантисса*, которая использует большую часть битов для описания точного значения, вне зависимости от модульной величины. И хотя точности действительных чисел, которую обеспечивают типы `float` и `double`, вполне достаточно для того, чтобы не беспокоиться об ограничениях, бывают случаи, когда специализированным приложениям необходим более прямой доступ к значениям с плавающей запятой.

Числа с плавающей запятой по определению обладают высокой точностью и большим диапазоном. Даже самый мелкий действительный тип в языке Java, `float`, может представлять в буквальном смысле астрономические величины, начиная с отрицательного значения 10^{-45} и заканчивая положительным 10^{38}. Это достигается за счет того, что мантисса действительного значения представляет фиксированное количество «цифр», а экспонента говорит нам о том, куда нужно вставить запятую (если использовать термины из десятичного мира). С ростом модульной величины числа увеличивается и количество цифр слева от десятичной запятой, поэтому и «точность» сдвигается «влево». Благодаря этому числа с плавающей запятой могут очень точно (благодаря большому количеству разрядов) представлять мелкие значения, такие как число Пи, но в случае с большими числами (с миллиардами и триллионами) эти разряды будут заняты более важными цифрами. Таким образом, промежуток между двумя соседними числами, которые могут быть представлены в виде значения с плавающей запятой, растет по мере того, как эти числа становятся больше.

Некоторым программам важно знать об ограничениях. Поэтому класс `java.lang.Math` предоставляет несколько методов для выяснения точности типов `float` и `double`. Метод `Math.ulp()` извлекает «меру

наивысшей точности» (unit of least precision или ULP) для заданного действительного числа — то есть наименьшее значение, которое представлено битами мантиссы в их текущей экспоненте. Другими словами, метод `ulp()` — это примерное расстояние от числа с плавающей запятой до ближайшего значения (большего или меньшего), которое может быть представлено. Добавив к значению типа `float` половину от показателя ULP, мы не получим новое число. Добавив к значению число, которое находится между половиной и целым показателем ULP, мы получим значение плюс ULP. Метод `Math.nextUp()` облегчает эту задачу: он принимает значение типа `float` и добавляет к нему показатель ULP, возвращая следующее число, которое может быть представлено.

```
float trillionish = (float)1e12; // trillionish ~= 999 999 995 904
float ulp = Math.ulp( f ); // ulp = 65536
float next = Math.nextUp( f ); // next ~= 1000000061440
trillionish += 32767; // trillionish все еще ~= 999 999 995 904. Без
изменений!
```

Кроме того, класс `java.lang.Math` содержит метод `getExponent()`, который извлекает из действительного числа его экспоненциальную часть (а дальше уже можно определить мантиссу, воспользовавшись делением). Числа типа `float` и `double` тоже можно получить в виде обычных битов, если воспользоваться соответствующими методами `floatToIntBits()` и `doubleToRawLongBits()` из класса-обертки и вычленить нужный фрагмент самостоятельно (следуя стандарту IEEE).

Случайные числа

Чтобы генерировать случайные значения, вы можете использовать класс `java.util.Random`. Это генератор псевдослучайных чисел, который инициализируется при помощи 48-битного начального значения*. Так как это псевдослучайный алгоритм, вы будете получать одинаковые наборы результатов для одного и того же начального значения. Отправной точкой для конструктора по умолчанию, который занимается генерированием, является текущее время, но вы также можете передать ему собственное число:

```
long seed = mySeed;
Random rnums = new Random( seed );
```

* Генератор использует линейную конгруэнтную формулу. См. «Искусство программирования. Том 2. Получисленные алгоритмы» Дональда Кнута (Donald Knuth).

Получив генератор, вы можете запросить у него одно или несколько значений различных типов, используя методы, перечисленные в табл. 11.3.

Табл. 11.3. Методы для генерирования случайных чисел

Метод	Диапазон
`nextBoolean()`	`true` или `false`
`nextInt()`	От `-2147483648` до `2147483647`
`nextInt(int n)`	От `0` до `(n - 1)` включительно
`nextLong()`	От `-9223372036854775808` до `9223372036854775807`
`nextFloat()`	От `0,0` (включительно) до `1,0` (не включительно)
`nextDouble()`	От `0,0` (включительно) до `1,0` (не включительно)
`nextGaussian()`	Нормальное (гауссово) распределенное число типа `double` медианой `0,0` и стандартным отклонением `1,0`

По умолчанию значения распределяются равномерно. При помощи метода `nextGaussian()` вы можете получить гауссово распределение значений типа `double` (в виде колоколообразной кривой) с медианой 0,0 и стандартным отклонением 1,0. (Большая часть случайных природных явлений подчиняются не равномерному, а гауссовому распределению).

Статический метод `Math.random()` возвращает случайные значения типа `double`. При первом вызове он инициализирует частный генератор случайных чисел из класса `Math`, используя конструктор по умолчанию `Random`.

Таким образом, каждый вызов `Math.random()` эквивалентен вызову метода `nextDouble()` из генератора случайных чисел.

Дата и время

Работа с датой и временем без подходящих инструментов может превратиться в утомительную рутину. К счастью, платформа Java содержит три класса, которые берут на себя большую часть этой работы. Класс `java.util.Date` инкапсулирует момент времени. Класс `java.util.GregorianCalendar`, который является наследником абстрактного класса `java.util.Calendar`, обеспечивает переход между моментом времени и календарными единицами измерения, такими как месяц, день и год.

Третий класс, `java.text.DateFormat`, умеет генерировать и разбирать строковые представления даты и времени во многих языках*.

Разделение классов `Date` и `Calendar` — это то же самое, что иметь два разных класса для представления температуры и преобразования этой температуры в градусы Цельсия. Класс `Date` представляет абсолютное значение времени, которое определяется количеством миллисекунд, прошедших с определенного момента — полуночи 1 января 1970 года по Гринвичу (GMT). Такой же способ отсчета применяется в методе `System.currentTimeMillis()`. Класс `Calendar` инкапсулирует понятие момента времени и переносит его в более высокоуровневое (и запутанное) представление, оперируя такими терминами, как год, месяц, неделя, день; он также берет на себя нерегулярные явления, такие как високосный год. Мы можем не ограничиваться стандартным классом `GregorianCalendar` и создавать другие производные от класса `Calendar`, которые будут представлять время с учетом разных социальных и культурных традиций**, например `JulianCalendar` или `LunarCalendar` .

Работа с календарями

Стандартный конструктор класса `GregorianCalendar` создает календарь, инициализированный при помощи текущего времени в текущем часовом поясе:

```
GregorianCalendar now = new GregorianCalendar();
```

Но чтобы не беспокоиться о том, какая календарная система используется в этом столетии, мы можем оставить выбор подходящего календаря классу `Calendar`:

```
Calendar now = Calendar.getInstance();
```

В любом случае вся реальная работа происходит в методах `set()` и `get()` класса `Calendar`. Чтобы ссылаться на поля и значения календаря, эти методы используют статические идентификаторы. Например:

```
Calendar birthday = Calendar.getInstance();
birthday.set( Calendar.YEAR, 1972 );
```

---

* До выхода платформы Java 1.1 класс `Date` тоже имел несколько методов для работы с календарем. К этому моменту большинство из них признаны устаревшими. На сегодня единственной функцией класса `Date` является представление момента времени.

** На самом деле класс `GregorianCalendar` из состава платформы Java, поддерживает как юлианский, так и григорианский календари с возможностью переключения даты.

```
birthday.set( Calendar.MONTH, Calendar.MAY );
birthday.set( Calendar.DATE, 20 );
```

Здесь мы задали в календаре значения для года, месяца и дня, изменяя внутренний экземпляр `Date` из объекта `Calendar`. Все поля, которые мы не указали, остались в том виде, в котором они были инициализированы (храня текущие дату и время на момент создания объекта). В данном случае мы не задали дату и время полностью; мы просто перезаписали отдельные поля в календаре.

Класс `Calendar` содержит идентификаторы для всех стандартных полей, связанных с датой и временем, а также значения вроде дня недели или номера месяца в году. В следующем списке представлены самые популярные идентификаторы:

- `YEAR, MONTH`
- `WEEK_OF_YEAR, WEEK_OF_MONTH`
- `DATE, DAY_OF_YEAR, DAY_OF_MONTH, DAY_OF_WEEK`
- `HOUR, HOUR_OF_DAY, AM_PM`
- `MINUTE, SECOND, MILLISECOND`
- `ZONE_OFFSET, DST_OFFSET`

Идентификаторы `DATE` и `DAY_OF_MONTH` являются синонимами. Значение `HOUR` имеет 12-часовой формат и может использоваться в сочетании с `AM_PM`. Назначение следующих идентификаторов должно быть понятно из их названий:

- `SUNDAY, MONDAY, TUESDAY...` — дни недели
- `JANUARY, FEBRUARY, MARCH...` — месяцы
- `AM, PM` — до полудня и после полудня

Помимо метода `set()`, который изменяет значения полей, класс `Calendar` предлагает два дополнительных метода для арифметических операций с датами — `add()` и `roll()`. При помощи метода `add()` вы можете легко перемещаться по календарю в обоих направлениях, используя любые единицы времени, для этого вам не придется вычислять значения других полей. К примеру, мы можем прокрутить наш календарь на четыре недели вперед:

```
   Calendar cal = Calendar.getInstance();
System.out.println( cal.getTime() );
// Thu Nov 04 16:39:06 EEST 2012
cal.add( Calendar.WEEK_OF_YEAR, 4 );
```

```
System.out.println( cal.getTime() );
// Thu Dec 02 16:39:06 EEST 2012
```

По умолчанию вывод даты и времени выполняется в американском формате. Чтобы эти значения выводились на русском языке, вам необходимо использовать классы `Locale` и `DateFormat`, речь о которых пойдет в следующих разделах.

Метод `roll()`, напротив, не заменяет содержимое полей календаря, а корректирует уже имеющиеся значения. Чтобы получить дополнительную информацию о добавлении и вычитании временных интервалов при помощи метода `add()`, вы можете ознакомиться с примером из главы 17, в котором идет речь о числовом поле `JSpinner`.

И, наконец, вы всегда можете получить внутреннее значение `Date` из объекта `Calendar` или же заново инициализировать календарь при помощи собственного экземпляра `Date`, используя методы `getTime()` и `setTime()`:

```
// Получаем абсолютное значение времени, на которое ссылается Calendar
Date date = calendar.getTime();
// Заново инициализируем календарь с помощью текущих даты и времени
Date now = new Date();
calendar.setTime( now );
```

Часовые пояса

Экземпляр класса `TimeZone` представляет часовой пояс и хранит информацию о переходах на зимнее и летнее время в заданной местности. Вы можете создавать часовые пояса на основе строк, применяя несколько разных форматов. Самый распространенный подход заключается в использовании смещения относительно GMT, но поддерживается также множество удобочитаемых вариантов (их полный список можно получить с помощью метода `TimeZone.getAvailableIDs()`).

```
TimeZone.getTimeZone("US/Central"); // CST (Центральное стандартное время
в США)
TimeZone.getTimeZone("GMT-06"); // CST
TimeZone.getTimeZone("America/Chicago"); // CST
```

Объект `Calendar` получает стандартный часовой пояс от платформы, на которой он был создан. Вы можете задавать разные часовые пояса при помощи метода `setTimeZone()`:

```
GregorianCalendar smokey = new GregorianCalendar();
smokey.setTimeZone( TimeZone.getTimeZone("US/Mountain") );
```

Важно, чтобы вы правильно воспринимали даты и часовые пояса. Запомните: объект `Date` — это абсолютное значение времени, а экземпляр `Calendar` переводит его в локализованные поля, которые могут зависеть от вашего местоположения. По большому счету, нет никакого смысла говорить о дате «14 сентября 2013 г.», не упоминая часовой пояс, так как в любой момент времени на нашей планете «сегодня» может означать разные календарные дни. Даже запись «14 сентября 2013 г. 19:05:37» будет неоднозначной, ведь эта комбинация полей календаря и времени проявится 24 раза на протяжении одного дня, по мере вращения земного шара (см. рис. 11.1).

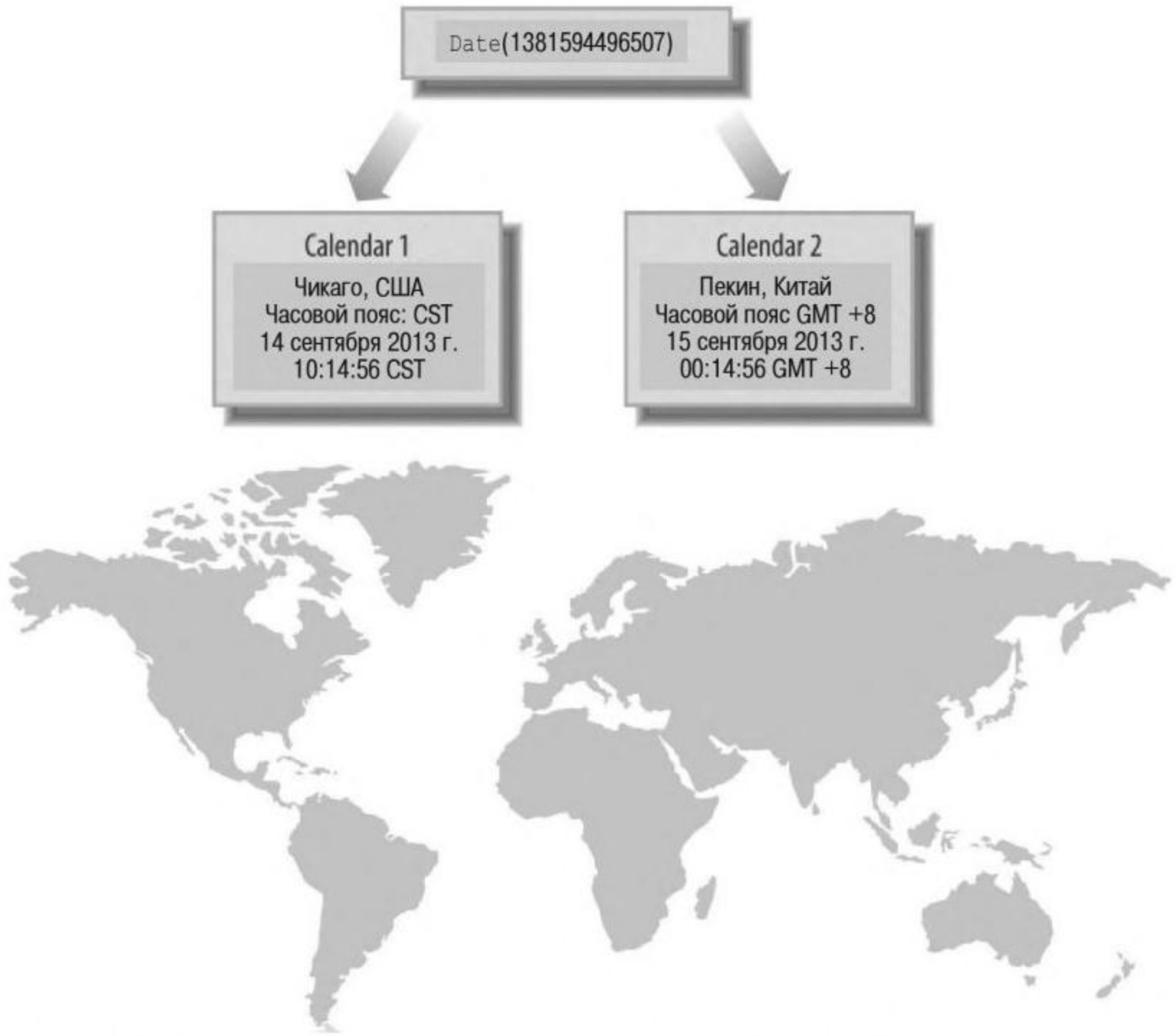

Рис. 11.1. Абсолютное временное значение переводится объектами `Calendar` в локальные дату и время

Только полное сочетание даты, времени и часового пояса может определить абсолютное значение времени — например, «14 сентября

2013 г. 19:05:37 EEST». Поэтому важно помнить, что класс `Calendar` присваивает всем этим полям значения по умолчанию, даже если вы сами их не трогаете.

В следующем примере выводится день недели для одного и того же объекта `Date`, но для разных часовых поясов:

```
Date date = new Date(); // момент времени
TimeZone CST = TimeZone.getTimeZone( "America/Chicago" );
Calendar usa = Calendar.getInstance( CST );
usa.setTime( date );
System.out.println( usa.get( Calendar.DAY_OF_WEEK ) ); // 1
TimeZone GMT8 = TimeZone.getTimeZone( "GMT+08"); // Пекин
Calendar china = Calendar.getInstance( GMT8 );
china.setTime( date );
System.out.println( china.get( Calendar.DAY_OF_WEEK ) ); // 2
```

В этом примере мы также могли бы легко изменить часовой пояс в календаре `usa` при помощи метода `setTimeZone()`. В отличие от действия методов `set()`, задание часового пояса не влияет на исходное значение `Date`, принадлежащее календарю, а всего лишь изменяет интерпретацию содержимого его полей.

Смысл объекта `Date` и его связь с классом `Calendar` становятся особенно важными при работе с API-интерфейсами наподобие баз данных, которые формируют значение на основе не до конца заполненных полей с датой и временем. Вполне возможно, что когда-нибудь вам придется отправлять ваш объект `Date` из клиентского приложения на сервер, который находится на другом конце планеты, и тогда вы можете удивиться тому, что поля в календаре изменились. В такой ситуации необходимо преобразовывать дату при помощи объектов `Calendar` путем добавления или вычитания из даты реальных значений времени, избегая «ручной подгонки».

Объект `Locale`

Сейчас у вас уже не должно оставаться никаких сомнений относительно того, что `Calendar` — это не просто причудливая надстройка над объектом `Date`, а нечто среднее между хронометром и приспособлением для форматирования времени. Это подтверждается и тем фактом, что класс `Calendar` учитывает региональные особенности. Помимо поддержки часовых поясов он содержит объект `Locale`, который отвечает

за такие условности, как определение начала и конца недели. Вы можете изменять региональные настройки при помощи метода `setLocale()`. Однако за большинство особенностей, присущих разным местностям, отвечает класс `DateFormat`, о котором мы поговорим ниже.

Разбор и форматирование даты при помощи класса `DateFormat`

Как можно догадаться из названия, класс `DateFormat` занимается форматированием объектов `Date`, не имея никакого отношения к классу `Calendar`. Поэтому первое, что нужно сделать для форматирования даты и времени, это получить из объекта `Calendar` значение типа `Date`. Делается это при помощи метода `getTime()`:

```
Date birthDate = calendar.getTime();
```

Чтобы представить дату и время в виде строки, создайте объект `DateFormat` и примените его метод `format()` к значению типа `Date`. Как и в случае с объектом `NumberFormat`, который мы рассматривали в предыдущей главе, экземпляр `DateFormat` сам по себе является абстрактным, но он содержит несколько статических («фабричных») методов, которые возвращают интересующие нас экземпляры производных классов. Чтобы получить стандартную разновидность `DateFormat`, достаточно просто вызвать метод `getInstance()`:

```
DateFormat simple = DateFormat.getInstance();
String now = simple.format( new Date() ); // 15.09.13 13:20
```

При помощи фабричных методов `getDateInstance()`, `getTimeInstance()` и `getDateTimeInstance()` вы можете сгенерировать строку с датой, временем или и с тем, и с другим. Аргументы для этих методов описывают уровень детализации, которую вам хотелось бы получить. За это отвечают четыре константы, которые содержатся в классе `DateFormat`: `SHORT`, `MEDIUM`, `LONG` и `FULL`. Есть также константа `DEFAULT`, которая означает то же, что и `MEDIUM`. В следующем коде создается три экземпляра `DateFormat`: один для даты, другой для времени и еще один для совместного форматирования даты и времени. В метод `getDateTimeInstance()` необходимо передать два аргумента, которые задают формат отдельно для даты и времени:

```
// 12-Apr-06
DateFormat df = DateFormat.getDateInstance(DateFormat.DEFAULT);
```

```
// 9:18:27 AM
DateFormat tf = DateFormat.getTimeInstance(DateFormat.DEFAULT);
// Wednesday, April 12, 2006 9:18:27 o'clock AM EDT
DateFormat dtf =
   DateFormat.getDateTimeInstance( DateFormat.FULL, DateFormat.FULL );
```

Здесь мы показываем только то, как создаются объекты `DateFormat`. Чтобы, собственно, сгенерировать строку из даты, вам нужно будет вызвать из этих объектов метод `format()`, передавая ему в качестве аргумента экземпляр `Date`.

Форматирование даты и времени для разных стран тоже не составляет труда. Перегруженные фабричные методы принимают аргумент `Locale`:

```
// 15.09.2013
DateFormat df =
   DateFormat.getDateInstance( DateFormat.DEFAULT, new Locale("ru") );
// 16:19:01
DateFormat tf =
   DateFormat.getTimeInstance( DateFormat.DEFAULT, Locale.GERMANY );
// domenica 15 settembre 2013 16.19.01 EEST
DateFormat dtf =
   DateFormat.getDateTimeInstance(
      DateFormat.FULL, DateFormat.FULL, Locale.ITALY );
```

Вы могли заметить, что для России формат указывается немного не так, как для других стран. Дело в том, что не во всех версиях платформы Java есть соответствующая константа, поэтому вместо того, чтобы писать `Locale.RUSSIA`, лучше использовать код страны (ru).

Для разбора строки, которая представляет собой дату, используется метод `parse()` из класса `DateFormat`. В качестве результата возвращается объект `Date`. Алгоритмы разбора являются довольно строгими, поэтому в качестве входных данных лучше использовать строки того формата, который поддерживает класс `DateFormat`. Если метод `parse()` не понимает строку, которую вы ему передали, он генерирует исключение `ParseException`. Ниже представлено несколько вызовов этого метода, и только последний из них завершится неудачно (формат `LONG` требует наличия часового пояса, который мы не указали). Время от времени метод `parse()` генерирует и другие исключения; чтобы охватить их все, вам также нужно обрабатывать экземпляры `NullPointerException` и `StringIndexOutOfBoundsException`.

```
try {
   Date d;
   DateFormat df;
   df = DateFormat.getDateTimeInstance(
            DateFormat.FULL, DateFormat.FULL);
   d = df.parse("15 сентября 2013 г. 16:45:41 EEST");
   df = DateFormat.getDateTimeInstance(
            DateFormat.MEDIUM, DateFormat.MEDIUM);
   d = df.parse("15.09.2013 16:45:41");
   df = DateFormat.getDateTimeInstance(
            DateFormat.LONG, DateFormat.LONG);
   d = df.parse("15 сентября 2013 г. 16:45:41 EEST");
   // генерирует исключение ParseException
   // недостаточный уровень детализации
   d = df.parse("15 сентября 2013 г. 16:45:41");
}
catch (Exception e) { ... }
```

Форматирование даты и времени в стиле метода `printf`

Форматирование в стиле метода `printf`, рассмотренное в главе 10, позволяет отображать дату и время в виде совершенно произвольных строк, и для этого не нужно получать доступ к отдельным полям при помощи методов из класса `Calendar`.

В строках форматирования для даты и времени всегда используется условный символ `t` (или `T`), за которым идет суффикс, обозначающий сам формат вывода отдельных полей. Например, строка форматирования `%tc` превращает аргумент `Date` в строковой эквивалент значения, которое можно получить при помощи стандартного метода `toString()` из того же объекта `Date`:

```
System.out.printf( "Дата: %tc\n", new Date() );
// Дата: Sun Sep 15 16:48:01 EEST 2013
```

Единственная разница между условными символами `t` и `T` заключается в том, что второй из них переводит весь вывод в верхний регистр — это стандартное поведение для условных символов в языке Java. При форматировании даты и времени учитываются региональные особенности, в том числе названия дней и месяцев, а также традиционные для США идентификаторы A.M./P.M. Чтобы отформатировать значение

Date для конкретного языка, просто передайте в качестве первого аргумента объект Locale:

```
System.out.printf( new Locale("ru"), "Дата: %tc\n", new Date() );
// Дата: Вс сен 15 16:48:25 EEST 2013
```

Как можно видеть в следующей таблице, существует несколько составных форматов, два из которых предназначены исключительно для даты, и три — только для времени. В описании строк форматирования в третьем столбце табл. 11.4 есть ссылки на отдельные условные символы, описанные в таблицах 11.5 и 11.6.

Табл. 11.4. Составные форматы даты и времени

Суффикс форматирования	Пример	Составляющие
c	Вс сен 15 21:47:25 EEST 2013	%ta %tb %td %tT %tZ %tY
D	09/15/13	%tm/%td/%ty
F	2013-09-15	%tY-%tm-%td
r	5:02:28 PM	%tI:%tM:%tS %Tp
R	17:03	%tH:%tM
T	17:03:27	%tH:%tM:%tS

В табл. 11.5 перечислены форматы доступа к компонентам даты.

Табл. 11.5. Специальные символы для даты

Суффикс форматирования	Примеры	Описание
a	Пн, Вт, Ср...	День недели (сокращенно)
A	понедельник, вторник...	День недели (в полном виде)
b	сен, окт, ноя...	Месяц (сокращенно)
B	Сентябрь, Октябрь...	Месяц (в полном виде)
Y	1999, 2013	Год (четырехзначный)
C	2013 = 20	Первые две цифры года
y	1999 = 99	Последние две цифры года
j	001 ... 366	День в году
m	01 ... 13	Месяц в году
d	01 ... 31	День месяца
e	1 ... 31	День месяца без ноля в начале

В табл. 11.6 перечислены форматы доступа к компонентам времени.

Табл. 11.6. Специальные символы для времени

Суффикс форматирования	Примеры	Описание
`H`	00 ... 23	24-часовой формат
`k`	0 ... 23	24-часовой формат без ноля в начале
`I`	01 ... 12	12-часовой формат
`l`	1 ... 12	12-часовой формат без ноля в начале
`M`	00 ... 59	Минуты
`S`	00 ... 60*	Секунды
`L`	000 ... 999	Миллисекунды
`p`	am, pm	Обозначения для утреннего и вечернего времени
`Z`	EEST, CST	Название часового пояса
`z`	+0400	Смещение часового пояса относительно GMT

* Второе число (60) является условным обозначением, которое используется для поддержки секунд координации.

Таймеры

Платформа Java содержит два полезных класса для выполнения кода в определенные моменты времени. К примеру, если вы создаете программный хронометр, вам может понадобиться обновлять его циферблат каждую секунду или воспроизводить звуковой сигнал в заранее определенный момент. Это можно сделать при помощи многопоточности и метода `Thread.sleep()`, но классы `java.util.Timer` и `java.util.TimerTask` созданы специально для решения таких задач.

Класс `Timer` работает как планировщик. Каждый его экземпляр содержит один поток, который работает в фоновом режиме и следит за временем; когда наступает определенный момент, он выполняет одну или несколько задач типа `TimerTasks`. Вы можете, например, запланировать однократное выполнение задачи в заданное время, как показано в следующем примере:

```
import java.util.*;
public class Y2K {
   public static void main(String[] args) {
```

```
        Timer timer = new Timer();
        TimerTask task = new TimerTask() {
           public void run() {
              System.out.println("С новым миллениумом!");
           }
        };
        Calendar cal = new GregorianCalendar( 2000, Calendar.JANUARY, 1 );
        timer.schedule( task, cal.getTime());
        }
}
```

Класс `TimerTask` реализует интерфейс `Runnable`. Чтобы создать задачу, вы можете просто наследовать `TimerTask`, определив для него метод `run()`. В этом примере мы создали простую производную от класса `TimerTask`, которая выводит сообщение через поток `System.out`. При помощи метода `schedule()` из объекта `Timer` мы сделали так, чтобы эта задача запустилась 1 января 2000 года. Если запланированное время уже прошло (как в нашем случае), задача выполнится немедленно.

Существуют и другие разновидности метода `schedule()`; вы можете запустить задачу только один раз или же выполнять ее регулярно с определенным интервалом. Периодические задачи можно описывать при помощи задержки или частоты. *Задержка* — это определенное количество времени, которое проходит между завершением задачи и началом ее нового выполнения. *Частота* определяет фиксированный временной интервал выполнения задачи. Разница между этими подходами начинает проявляться в тех случаях, когда время выполнения задачи становится сравнимым с длиной интервала. Помните, что задачи выполняются в едином потоке планирования объекта `Timer`. Если одна из них работает слишком долго, то запуск остальных может быть задержан — в этом случае они будут стартовать сразу же, как только поток станет доступным.

Ниже представлен пример кода для ежесекундного обновления циферблата часов:

```
Timer timer = new Timer();
TimerTask task = new TimerTask() {
   public void run() {
      repaint(); // обновление циферблата часов
   }
};
timer.scheduleAtFixedRate( task, 0, 1000 );
```

Если задача типа `TimerTask` еще не начала выполняться, она может быть отменена с помощью метода `cancel()`.

Коллекции

В любой области программирования *коллекции* считаются фундаментальными структурами данных. Каждый раз, когда нам нужно сослаться на группу объектов, мы имеем дело с некоего рода коллекцией. Простейшие коллекции в виде массивов поддерживаются платформой Java на уровне языка. Но массивы статичны, они имеют фиксированную длину и поэтому плохо подходят для групп объектов, которые расширяются и сужаются на протяжении работы программы. Они также недостаточно хорошо представляют взаимосвязь между объектами. На раннем этапе своего развития платформа Java поддерживала только два простых класса, которые предназначались для этих целей: `java.util.Vector`, представляющий динамический список объектов, и `java.util.Hashtable`, хранящий ассоциативный массив элементов вида «ключ/значение». В наши дни платформа Java придерживается более комплексного подхода по отношению к коллекциям, имея в своем составе технологию Collections Framework. Старые классы никуда не делись, постепенно став частью платформы (с некоторыми оговорками), но сейчас они, как правило, не используются.

Несмотря на свою концептуальную простоту, коллекции являются одним из наиболее мощных компонентов в любом языке программирования. Они реализуют структуры данных, которые лежат в основе решения сложных задач. Значительная часть информатики как науки посвящена описанию самых эффективных способов реализации определенных видов алгоритмов, которые работают с коллекциями. Имея в своем распоряжении эти инструменты и понимая, как их использовать, вы можете сделать свой код куда более лаконичным и быстрым. Это также может уберечь вас от изобретения очередного «велосипеда».

До выхода платформы Java 5 технология Collections Framework имела два больших недостатка. Во-первых, в виду отсутствия обобщенных типов, коллекции должны были быть нетипизированными, и вместо реальных типов, таких как `Date` или `String`, работать только с анонимными объектами (`Object`). Из-за этого при каждом извлечении объекта из коллекции приходилось выполнять приведение типов. Это был плевок в сторону безопасной типизации на этапе компилирования, которая

лежит в основе Java. Но на практике этот подход не столько приводил к проблемам, сколько был громоздким и трудоемким. Второй недостаток заключался в том, что, исходя из практических соображений, коллекции могли работать только с объектами, но не с простыми типами. Это означало, что каждый раз, когда в коллекцию нужно было поместить число или другое простое значение, его приходилось предварительно заключать в класс-обертку, а при извлечении наоборот — распаковывать. Сочетание этих двух факторов делало код, работающий с коллекциями, менее понятным и более опасным при выполнении.

Но все это ушло в прошлое, когда появились обобщенные типы и автоматическая упаковка простых значений.

Прежде всего, благодаря наличию обобщенных типов, которые были описаны в главе 8, в распоряжении программиста появились коллекции с поддержкой настоящей безопасной типизации. Появление автоматической упаковки/распаковки на уровне коллекций позволило относиться к простым типам так же, как к обычным объектам. Сочетание этих двух нововведений может не только значительно уменьшить объем вашего кода, но и сделать его безопасным. Как мы позже убедимся, вышеупомянутые возможности присутствуют во всех классах коллекций.

Технология Collections Framework основана на нескольких интерфейсах из пакета `java.util`, которые разделены на две ветви. Первая наследуется от интерфейса `Collection`. Он (и его производные) представляют собой контейнер, который хранит другие объекты. Вторая ветвь основана на интерфейсе `Map`, который описывает набор элементов вида «ключ/значение», где ключ используется для эффективного извлечения объектов.

Интерфейс `Collection`

Прародителем всех коллекций является интерфейс с говорящим названием `Collection`. Он выполняет функции контейнера, который хранит внутри себя другие объекты, то есть свои *элементы*. Он не описывает то, как именно эти объекты хранятся; к примеру, он не может вам сказать, допустимо ли дублирование объектов и должны ли элементы как-то упорядочиваться. Такие подробности отданы на откуп дочерним интерфейсам. Тем не менее интерфейс `Collection` определяет некоторые базовые операции, которые являются общими для всех коллекций:

```
public boolean add( element )
```

Этот метод добавляет в коллекцию указанный элемент. Если добавление прошло успешно, метод вернет `true`. Если объект уже находится в коллекции и дубликаты не допускаются, будет возвращено значение `false`. Не стоит также забывать, что некоторые коллекции поддерживают только чтение, и если вы вызовете этот метод, они сгенерируют исключение. `UnsupportedOperationException`.

```
public boolean remove( element )
```

Этот метод удаляет указанный объект из коллекции. Как и в случае с методом `add()`, он возвращает `true`, если удаление прошло успешно. Если такого объекта в коллекции нет, будет возвращено значение `false`. Коллекции, доступные только для чтения, в случае вызова этого метода сгенерируют исключение. `UnsupportedOperationException`.

```
public boolean contains( element )
```

Этот метод возвращает `true`, если коллекция содержит указанный элемент.

```
public int size()
```

Используйте этот метод для того, чтобы узнать количество элементов в коллекции.

```
public boolean isEmpty()
```

Этот метод возвращает `true`, если коллекция не содержит элементов.

```
public Iterator iterator()
```

Этот метод предназначен для перебора всех элементов коллекции. Он возвращает объект `Iterator` (итератор), с помощью которого можно пройтись по каждому элементу. Мы обсудим итераторы в следующем разделе.

Кроме того, существуют методы `addAll()`, `removeAll()` и `containsAll()`, которые принимают второй объект `Collection` и, соответственно, добавляют, удаляют или проверяют на наличие все его элементы.

Обобщенные типы и коллекции

Коллекции можно использовать совместно с обобщенными типами; при этом выполняется параметризация, то есть задание типа элементов, которые будут храниться. Благодаря этому универсальное хранилище

для «всего на свете» можно превратить в коллекцию какого-то определенного типа. На этапе компиляции параметр типа становится типом аргументов во всех методах интерфейса `Collection` (в нашем случае это методы `add()`, `remove()` и `contains()`, перечисленные выше). В качестве примера создадим коллекцию, которая работает с типом `Date`:

```
Collection<Date> dates = new ArrayList<Date>();
// Запись "= new ArrayList<>()" тоже будет работать
dates.add( new Date() );
// Ошибка; мы добавили строку там, где ожидается Date!!
dates.add( "foo" )
```

Класс `ArrayList` — это всего лишь одна из реализаций интерфейса `Collection`; мы поговорим о нем чуть позже. Стоит заметить, что мы объявили переменную `dates` с типом `Collection<Date>`, то есть коллекцию значений типа `Date`. Мы также выделили для нее соответствующий объект `ArrayList`. Наша коллекция была параметризирована при помощи типа `Date`, поэтому ее метод `add()` превратился в `add( Date date )`, и теперь любая попытка добавить в список значение другого типа приведет к ошибке еще на этапе компиляции.

При работе со старым кодом, который был написан до появления обобщенных типов, вы можете просто пропускать параметризацию, выполняя подходящие преобразования постфактум. Например:

```
Collection dates = new ArrayList();
// Тип не проверяется, предупреждение на этапе компиляции
dates.add( new Date() );
Date date = (Date)dates.get( 0 );
```

В этом случае на этапе компиляции мы получим предупреждение о том, что использование объекта `ArrayList` таким образом может быть небезопасно (необобщенная типизация).

В сущности, в этом и заключается роль компилятора при работе с обобщенными типами, и об этом мы уже упоминали ранее. На платформе Java версии 5 или выше подобное использование коллекций (или любых других обобщенных типов) будет вызывать предупреждения на этапе компиляции, сигнализирующие об отсутствии проверки. Это означает, что вы можете получить ошибку во время выполнения программы, если сделаете что-то не так. В этом примере ошибка сможет проявиться только после того, как кто-нибудь попытается извлечь объект из коллекции и привести его к ожидаемому типу.

Устаревший код и безопасная типизация на этапе выполнения

Безопасной типизации можно добиться даже при работе с устаревшим кодом, который был написан до выхода платформы Java 5 и не поддерживает обобщенные типы. Для этого типы коллекций необходимо проверять во время выполнения кода, используя методы-обертки, которые платформа Java предоставляет для всех реализаций `Collection`. Эти методы явно приводят значения к заданному типу, генерируя исключение `ClassCastException` при попытке добавления неподходящего элемента. Например:

```
List list = new ArrayList();
list = Collections.checkedList( list, Date.class );
list.add( new Date() );
// Генерируется исключение ClassCastException!
list.add( "foo" );
```

Здесь статический метод `Collections.checkedList()` создает для нашей коллекции `list` обертку, которая реализует все методы интерфейса `List`, следя за тем, чтобы в ней хранились только значения типа `Date`. В качестве второго аргумента выступает значение `Date.class`, которое ссылается на класс `Date`. Благодаря этому обертка понимает, какой тип мы хотим применить. Подобные методы с префиксом `checked` существуют для всех основных интерфейсов, которые описывают коллекции, включая `Collection`, `List`, `Set` и `Map`.

Преобразование коллекций в массивы и наоборот

Превратить коллекцию в массив довольно легко. В целях удобства были предусмотрены следующие методы, которые позволяют извлекать элементы из коллекции в виде массива:

```
public Object[] toArray()
public <E> E[] toArray( E[] a )
```

В первом случае возвращается обычный массив типа `Object`. Второй метод вносит больше конкретики, позволяя получить массив с элементами нужного нам типа. Массив достаточного размера сразу заполняется значениями. Если же он будет слишком коротким (например, нолевой длины), мы получим взамен новый массив *того же типа, но с необходимой длиной*. Поэтому вы можете просто передать пустой массив нужного вам типа, как показано ниже:

```
Collection<String> myCollection = ...;
String [] myStrings = myCollection.toArray( new String[0] );
```

(Этот прием немного неуклюжий, и было бы лучше, если бы язык Java позволял указывать тип явно, используя ссылку `Class`. Но по каким-то причинам этого делать нельзя). Можно зайти и с другой стороны, преобразовывая массив объектов в коллекцию `List`. Это делается при помощи статического метода `asList()` из класса `java.util.Arrays`:

```
String [] myStrings = ...;
List list = Arrays.asList( myStrings );
```

Итератор

Итератор — это объект, который позволяет перебирать последовательность значений. Подобную работу приходится выполнять настолько часто, что для нее придумали стандартный интерфейс — `java.util.Iterator`. Интерфейс `Iterator` содержит всего два главных метода:

```
public E next()
```

Этот метод возвращает из коллекции следующий элемент (обобщенного типа `E`).

```
public boolean hasNext()
```

Этот метод возвращает `true`, если вы еще не перебрали все элементы коллекции. Другими словами, он возвращает `true`, если вы можете получить следующий элемент при помощи метода `next()`.

На примере следующего кода показано, как при помощи объекта `Iterator` вывести каждый элемент коллекции:

```
public void printElements(Collection c, PrintStream out) {
   Iterator iterator = c.iterator();
   while ( iterator.hasNext() )
      out.println( iterator.next() );
}
```

Помимо методов для перебора объект `Iterator` дает возможность удалять элементы из коллекции:

```
public void remove()
```

Этот метод удаляет последний объект, который был возвращен из коллекции методом `next()`.

Не во всех итераторах реализован метод `remove()`. К примеру, нет никакого смысла удалять элементы из коллекции, которая поддерживает только чтение. Если удаление элементов запрещено, этот метод сгенерирует исключение `UnsupportedOperationException`. Вызвав его два раза подряд или до первого срабатывания метода `next()`, вы получите исключение `IllegalStateException`.

Перебор коллекций с помощью цикла `for`

Разновидность цикла `for`, описанная в главе 4, способна работать с любыми типами объектов `Collection`. Например, мы можем пройтись по всем элементам типа `Date`, хранящимся в коллекции, следующим образом:

```
Collection<Date> col = ...
for( Date date : col )
   System.out.println( date );
```

Такая стандартная для языка Java форма цикла `for` называется «улучшенной» (на противовес предопределенному циклу `for`, который поддерживает только числовые индексы). Она применима только к коллекциям типа `Collection`, но не к ассоциативным массивам `Map`. `Map` — это коллекция иного рода, которая на самом деле содержит два набора объектов (ключи и значения), поэтому при использовании такого цикла было бы не совсем понятно, чего именно вы хотите добиться.

Интерфейс `java.util.Enumeration`

До появления технологии Collections Framework существовал другой интерфейс итератора — `java.util.Enumeration`. Его методы имели чуть более длинные названия, такие как `nextElement()` или `hasMoreElements()`, но делали они то же самое. Многие новые классы предоставляют реализацию `Enumeration` там, где сейчас мог бы использоваться объект `Iterator`. Если вам не важна производительность, вы можете просто преобразовать интерфейс `Enumeration` в список `List`, используя вспомогательный метод из класса `java.util.Collections`:

```
Enumeration myEnumeartion = ...;
List list = Collections.list( myEnumeration );
```

Виды коллекций

У интерфейса `Collection` есть три наследника. Интерфейс `Set` представляет собой коллекцию, элементы которой не могут дублироваться.

`List` — это коллекция, чьи элементы размещаются в определенном порядке. Интерфейс `Queue` является буфером для объектов, чья главная идея заключается в том, что для обработки всегда выбирается элемент, который находится в самом начале.

Интерфейс `Set`

У интерфейса `Set` (набор) совсем нет методов, если не считать тех, что наследуются от `Collection`. Он просто следит за соблюдением правила о недопустимости дубликатов. Если вы попытаетесь добавить элемент, который уже существует, метод `add()` просто вернет `false`. Подмножество этого интерфейса под названием `SortedSet` хранит элементы в заданном порядке; фактически это отсортированный список, у которого не может быть повторений. Он также дополняет интерфейс `Set` методами `add()` и `remove()`. Вы можете извлечь из него подмножество элементов, используя методы `subSet()`, `headSet()` и `tailSet()`. Эти методы принимают один или два аргумента, которые обозначают границы подмножества. Методы `first()` и `last()` предоставляют доступ к первому и последнему объекту в списке, а `comparator()` используется для сравнения элементов (об этом чуть позже).

В платформе Java 7 появился интерфейс `NavigableSet`, который наследует `SortedSet`; вместе с ним появились методы, которые ищут в рамках отсортированного списка элемент, меньше всего отличающийся от заданного значения (в большую или меньшую сторону). Данный интерфейс можно легко реализовать при помощи таких методик, как *список с пропусками*, благодаря которым повышается скорость поиска упорядоченных элементов (платформа Java 7 предоставляет такую реализацию, о чем мы поговорим позже).

Интерфейс `List`

Еще одним наследником интерфейса `Collection` является `List` (список). `List` — это упорядоченная коллекция, похожая на массив, но с методами для изменения позиции элементов в списке:

```
public boolean add( E element )
```

Этот метод добавляет в конец списка указанный элемент.

```
public void add( int index , E element )
```

Этот метод вставляет заданный объект в определенную позицию в списке. Если номер позиции меньше ноля или больше длины списка,

генерируется исключение IndexOutOfBoundsExcep. Все элементы, начиная с того, который ранее находился в указанном месте, сдвигаются «вправо» на одну позицию (то есть их индекс увеличивается на 1).

```
public void remove( int index )
```

Этот метод удаляет элемент по заданному индексу. Все элементы, которые шли после него, сдвигаются на одну позицию назад.

```
public E get( int index )
```

Этот метод возвращает элемент, который находится по заданному индексу.

```
public Object set( int index , E element )
```

Этот метод вставляет указанный объект вместо элемента, который находится в заданной позиции. По указанному индексу должен находиться какой-то элемент, иначе будет сгенерировано исключение IndexOutOfBoundsException.

Тип E в этих методах обозначает параметризированный тип элемента в классе List. Collection, Set и List — все это интерфейсные типы. С их конкретными реализациями мы вскоре познакомимся.

Интерфейс Queue

Интерфейс Queue (очередь) — это коллекция, которая ведет себя, как буфер для элементов. Этот интерфейс соблюдает порядок добавления значений, а также отдельно выделяет элемент, который находится «первым в очереди». В зависимости от реализации очередь может строиться по принципам «первым пришел — первым ушел» (first in, first out — FIFO) или «последним пришел — первым ушел» (last in, first out — LIFO):

```
public boolean offer( E element )
public boolean add( E element )
```

Метод offer() пытается поместить элемент в очередь и в случае успеха возвращает true. Разные виды интерфейса Queue могут иметь собственные ограничения относительно типов элементов (в том числе связанные с их объемом). Если элемент не может быть размещен внутри коллекции, метод offer() возвращает значение типа Boolean (в отличие от метода add(), унаследованного от интерфейса Collection, который в таком случае генерирует исключение).

```
public E poll()
public E remove()
```

Метод `poll()` убирает из очереди ее первый элемент и возвращает его в качестве результата. Он отличается от метода `remove()` из интерфейса `Collection` тем, что в случае, если коллекция пустая, возвращает `null`, а не генерирует исключение.

```
public E peek()
```

Этот метод возвращает из очереди ее первый элемент, не удаляя его оттуда. Если очередь пустая, возвращается `null`.

В платформе Java 7 появился новый интерфейс, `Deque`. Это двунаправленная очередь, которая поддерживает добавление, удаление и обращение к элементам с любого конца (сзади или спереди). У интерфейса есть свои версии методов для работы с очередью — `offer`, `poll` и `peek` — которые умеют работать как с первым, так и с последним элементом: `offerFirst()`, `pollFirst()`, `peekFirst()`, `offerLast()`, `pollLast()`, `peekLast()`. Стоит заметить, что этот интерфейс наследует `Queue`, поэтому он имеет тот же тип. Если вы будете вызывать из него методы `offer()`, `poll()` и `peek()`, он будет вести себя как очередь вида FIFO. В частности, метод `offer()` является аналогом `offerLast()`, а методы `poll()` и `peek()` эквивалентны, соответственно, методам `pollFirst()` и `peekFirst()`.

И, наконец, платформа Java содержит старый класс `Stack`, который ведет себя как очередь вида LIFO с операциями push (добавление) и pop (удаление). Но в целом интерфейс `Deque` является лучшим решением и должен рассматриваться как главный кандидат на замену очереди `Stack`. Просто используйте методы `addFirst()` и `pollFirst()` для операций push и pop соответственно.

Интерфейс `BlockingQueue`

Интерфейс `BlockingQueue` (блокирующая очередь) входит в состав пакета `java.util.concurrent`. Это расширение интерфейса `Queue`, предназначенное для очередей, у которых может быть фиксированный объем или другие ограничения, связанные со временем. Интерфейс `BlockingQueue` позволяет блокировать очередь в ожидании элемента, который можно добавить или извлечь. Он содержит собственные версии методов `offer()` и `poll()`, работа которых зависит от времени, а также блокирующие разновидности методов `take()` и `put()`:

```
public boolean offer( E element, long time, TimeUnit units )
```

Этот метод пытается разместить элемент в очереди, но, в отличие от своего аналога из интерфейса `Queue`, блокирует ее в ожидании свободного места (не дольше указанного времени).

```
public E poll( long time, timeUnit unit )
```

Этот метод пытается удалить элемент из начала очереди, но, в отличие от своего аналога из интерфейса `Queue`, блокирует ее, ожидая, пока элемент не станет доступным (не дольше указанного времени).

```
public E take()
```

Этот метод извлекает элемент из начала очереди, блокируя ее при необходимости, пока этот элемент не станет доступен.

```
public void put( E element )
```

Этот метод добавляет элемент в очередь, блокируя ее при необходимости, пока не появится свободное место.

```
public boolean add( E element )
```

Этот метод пытается мгновенно добавить элемент в очередь. В случае успеха он возвращает `true`. При нехватке свободного места генерируется исключение `IllegalStateException`. Этот метод может пригодиться в тех случаях, когда вы уверены, что очередь никогда не откажется принимать ваш элемент.

Интерфейс `Map`

Технология Collections Framework также содержит интерфейс `java.util.Map`, который представляет собой коллекцию элементов вида «ключ/значение». Ее еще часто называют *словарем* или *ассоциативным массивом*. Подобного рода коллекции сохраняют и извлекают элементы при помощи ключей; они отлично подходят для использования в таких инструментах, как кеши или минималистические базы данных. Сохраняя значение в ассоциативном массиве, вы связываете (ассоциируете) элемент с неким ключом. Когда вам нужно будет получить этот элемент, вы сможете извлечь его при помощи того же ключа.

Интерфейс `Map` параметризируется при помощи двух обобщенных типов: один для ключей, а другой для значений. В следующем примере кода используется класс `HashMap` — эффективная реализация ассоциативного массива (которую мы обсудим позже):

```
Map<String, Date> dateMap = new HashMap<String, Date>();
dateMap.put( "сегодня", new Date() );
Date today = dateMap.get( "сегодня" );
```

Раньше в ассоциативных массивах между собой связывались объекты `Object`; это требовало соответствующего приведения типов при извлечении элементов.

Основные операции с интерфейсом `Map` выполняются довольно просто. В следующих методах `K` обозначает тип ключа, а `V` — тип значения:

```
public V put( K key , V value )
```

Этот метод добавляет в ассоциативный массив заданную пару «ключ/значение». Если в коллекции уже содержится значение с таким ключом, оно будет обновлено и возвращено в качестве результата.

```
public V get( K key )
```

Этот метод извлекает из ассоциативного массива значение, связанное с заданным ключом.

```
public V remove( K key )
```

Этот метод удаляет из ассоциативного массива значение, связанное с заданным ключом, и возвращает его в качестве результата.

```
public int size()
```

Используйте этот метод, чтобы узнать количество пар «ключ/значение», хранящихся в ассоциативном массиве.

Вы можете извлечь из ассоциативного массива все ключи или значения:

```
public Set keySet()
```

Этот метод возвращает объект `Set`, который содержит все ключи коллекции.

```
public Collection values()
```

Используйте этот метод, чтобы извлечь все значения ассоциативного массива.

Возвращаемая коллекция может содержать повторяющиеся элементы.

У ассоциативного массива Map есть дочерний интерфейс, SortedMap, который поддерживает элементы в определенном порядке, исходя из значений ключей. Он предоставляет методы subMap(), headMap() и tailMap(), которые позволяют извлекать подмножества отсортированного ассоциативного массива. Как и интерфейс SortedSet, он предоставляет метод comparator(), возвращающий объект, который определяет то, как именно упорядочены ключи коллекции (об этом мы поговорим позже). В платформе Java 7 появился интерфейс NavigableMap, который по своим функциям напоминает NavigableSet, а именно: он содержит методы для поиска элемента, находящегося в упорядоченном ассоциативном массиве, и меньше всего отличается от заданного значения (в большую или меньшую сторону).

И, наконец, мы должны кое-что прояснить: хотя интерфейс Map и считается коллекцией, его тип не происходит от Collection (Map не наследует интерфейс Collection). Вам, наверное, интересно почему. В контексте ассоциативного массива имели бы смысл все методы интерфейса Collection, за исключением одного — iterator(). Напомним, что интерфейс Map имеет два набора объектов: ключи и значения. И для каждого из этих наборов нужен свой отдельный итератор. Вот почему Map не реализует интерфейс Collection.

Еще одно замечание по поводу ассоциативных массивов: некоторые реализации (включая стандартный для платформы Java класс HashMap) позволяют использовать null в качестве ключей и значений, однако не стоит всегда на это полагаться.

Интерфейс ConcurrentMap

Интерфейс ConcurrentMap входит в состав пакета java.util.concurrent. Он наследует базовый интерфейс Map, оснащая его атомарными операциями добавления, удаления и замены, которые могут пригодиться при параллельных вычислениях:

```
public V putIfAbsent( K key, V value )
```

Этот метод связывает значение с ключом, но только в том случае, если такой ключ еще не используется. Если ключ уже существует, то ничего не произойдет. В противном случае ключ будет создан. Итоговое значение (существующее или новое) будет возвращено в качестве результата.

```
public boolean remove( Object key, Object value )
```

Этот метод удаляет связанные между собой ключ и значение, только если они совпадают с предоставленными аргументами. Если пара была удалена, возвращается `true`, если нет — `false`.

```
public boolean replace( K key, V existingValue, V newValue)
```

Этот метод заменяет значение, связанное с ключом, но только если оно равно аргументу `existingValue`. Если замена была выполнена, возвращается `true`.

```
public boolean replace( K key, V value )
```

Этот метод заменяет значение, но только если оно уже существует для заданного ключа.

Реализации коллекций

До этого момента мы говорили только об интерфейсах. Но чтобы получить экземпляр коллекции, вам нужна ее конкретная реализация. Естественно, технология Collections Framework включает в себя удобные реализации всех интерфейсов коллекций. Для некоторых интерфейсов их может быть сразу несколько на выбор. Чтобы понять разницу между всеми этими реализациями, желательно иметь представление о нескольких наиболее распространенных структурах данных, которые встречаются во всех языках программирования; речь идет о массивах, связных списках, деревьях и хеш-таблицах. Этой теме посвящено множество книг, поэтому мы не будем обременять вас скучными подробностями. Мы коротко пройдемся по основным моментам, а потом приступим к обсуждению реализаций, представленных на платформе Java. Но сначала стоит подчеркнуть, что разница между этими реализациями становится ощутимой только при работе с большим количеством элементов или в условиях, когда время начинает играть крайне важную роль. По большому счету все они работают достаточно хорошо и являются взаимозаменяемыми.

Массивы

Вполне очевидно, что старый добрый Java-*массив*, показанный на рис. 11.2, хорошо подойдет для хранения коллекций с упорядоченными элементами. Однако, как мы уже упоминали ранее, у массивов есть одно ограничение: они не могут расширяться. Чтобы поддерживать настоящую коллекцию, массив должен уметь увеличивать свой объем, а для

этого его содержимое придется копировать в массив большего размера. Еще одна проблема, которая возникает при использовании массива в качестве списка, заключается в том, что для вставки элемента в определенную позицию или изъятия его оттуда опять нужно будет заниматься копированием больших участков массива, что очень неэкономно.



Рис. 11.2. Структура массива

По этой причине считается, что время доступа к элементам массива является *постоянным*, но время на добавление и удаление растет *линейно*. Это означает, что в целом время на чтение элементов остается примерно одинаковым, вне зависимости от того, сколько их хранится в массиве (это достигается за счет того, что массивы полностью индексируются); однако по мере увеличения количества элементов время на их добавление и удаление начинает расти прямо пропорционально, что выливается в линейную зависимость. Но это еще не худший вариант, ведь иногда встречаются алгоритмы с экспоненциальным ростом, из-за которого их невозможно использовать для большого количества элементов. Если не считать редких исключений, о которых будет сказано отдельно, все реализации коллекций на платформе Java обеспечивают в худшем случае линейное увеличение времени.

Массивы могут пригодиться в тех случаях, когда вам нужно преимущественно читать содержимое коллекции или добавлять элементы исключительно в ее конец.

Связные списки

Связной список, представленный на рис. 11.3, хранит элементы в виде цепочки *узлов*, каждый из которых ссылается на предыдущий и следующий узел (если таковые имеются). Таким образом, связной список формирует упорядоченную коллекцию, которая выглядит как массив. Но, в отличие от массива, у связного списка внутри нет никакой «магии», поэтому, чтобы получить из него элемент, приходится последовательно перебирать все значения (с начала или конца), пока не будет достигнута нужная позиция. Как вы уже, наверное, догадались, эта операция имеет линейную зависимость от времени — чем больше элементов в списке, тем дольше она выполняется. С другой стороны, когда вы

доберетесь до нужного места, добавление и удаление элементов станет проще некуда: достаточно поменять ссылки — и готово. Это означает, что в начале и конце связного списка операции добавления и удаления должны всегда занимать одно и то же время.



Рис. 11.3. Структура связного списка

Связные списки полезны в ситуациях, когда нужно выполнять много добавлений или удалений. Интересной разновидностью этой базовой структуры данных является *список с пропусками*; в сущности, это связной список, который вместо связей между соседними узлами в цепочке поддерживает иерархию ссылок, охватывающих постоянно расширяющийся диапазон элементов. Идея в том, что для перехода в середину списка можно воспользоваться одним из «быстрых указателей»; они переносят вас примерно туда, куда вам нужно, после чего, при необходимости, вы можете перемещаться вперед или назад с нужной вам точностью, используя иерархию указателей. В платформе Java 7 была добавлена поддержка списков с пропусками, которые реализуют интерфейсы `NavigableMap` и `NavigableSet` из пакета `java.util.concurrent`, и предназначены для параллельных вычислений.

Деревья

Деревья, как и связные списки, хранят свои элементы в виде узлов, которые ссылаются на своих соседей. Но, как понятно из названия, они имеют нелинейную структуру; их элементы собираются в каскады «ветвей», как в генеалогическом древе. Сильная сторона древовидных структур заключается в сортировке и поиске элементов, размещенных в определенном порядке. *Двоичное дерево поиска*, представленное на рис. 11.4, организовывает свое содержимое таким образом, чтобы дочерние элементы разделяли между собой диапазон значений. Один дочерний элемент хранит значения, которые больше, чем у заданного узла, а другой те, что меньше. Если рекурсивно применить эту информацию к достаточно «сбалансированному» дереву, можно быстро найти любое значение. Зависимость времени поиска от количества элементов является логарифмической и описывается формулой *log(n)*; это намного лучше линейной зависимости, которая, в сущности, означает бесхитростный перебор всех элементов подряд.

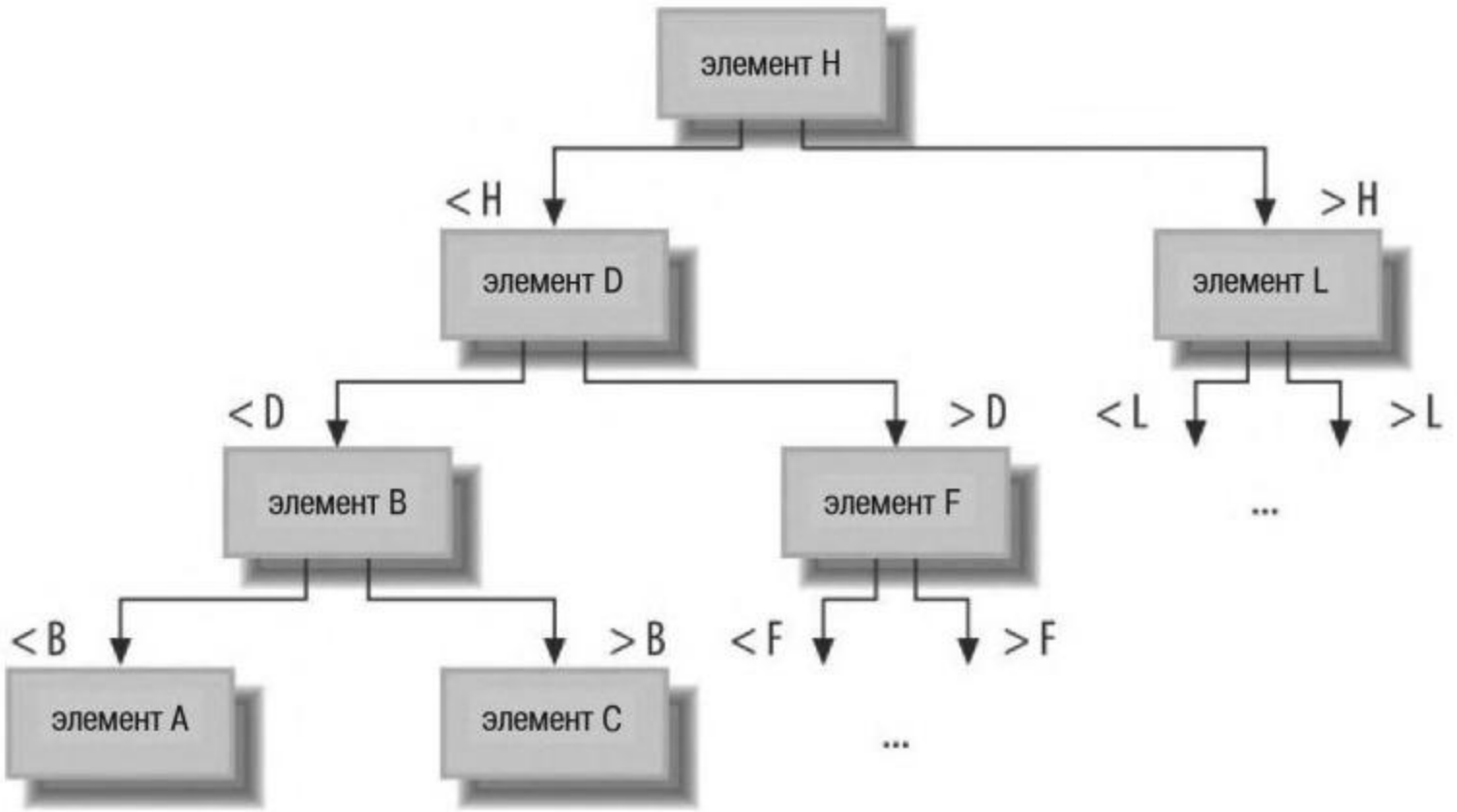

Рис. 11.4. Древовидная структура

Деревья подходят для хранения больших коллекций отсортированных элементов и поиска по ним. Похожую концепцию, которая годится для работы с редко изменяющимися данными, предлагает обычный упорядоченный массив в сочетании алгоритмом двоичного поиска. Двоичный поиск позволяет (экспоненциально) уменьшить количество переходов для получения примерной позиции элемента в упорядоченном массиве; чтобы получить точную позицию, вам придется рассчитать свой следующий переход исходя из того, насколько вы «промахнулись». Класс `Arrays` из состава платформы Java содержит несколько методов `binarySearch()`, которые поддерживают разные виды массивов.

Хеш-таблицы

Хеш-таблицы (или просто *хеши*, как их еще называют) — это странные магические создания. Они применяют к своим ключам математический алгоритм хеширования, распределяя значения по набору «сегментов». Этот алгоритм обеспечивает максимально равномерное (случайное) распределение элементов. Получение элемента по его ключу происходит путем поиска нужного сегмента. Операция хеширования имеет высокую скорость и может поддерживать большое количество сегментов; на практике приходится перебирать всего несколько значений, поэтому доступ выполняется очень быстро. Как уже было сказано в главе 7, каждый объект в языке Java имеет свой хеш, который определяется методом `hashCode()`. Подробнее о хеш-кодах и значениях ключей для ассоциативных массивов мы поговорим позже в этой главе.

Производительность хеш-таблицы зависит от многих факторов, включая сложность алгоритма хеширования, который реализуют ее элементы (см. рис. 11.5). В целом, если выбрать удачную реализацию хеш-функции, класс `HashMap` из состава платформы Java будет иметь постоянное время записи и доступа к элементам. Хеш-таблицы хорошо справляются с неупорядоченными коллекциями.

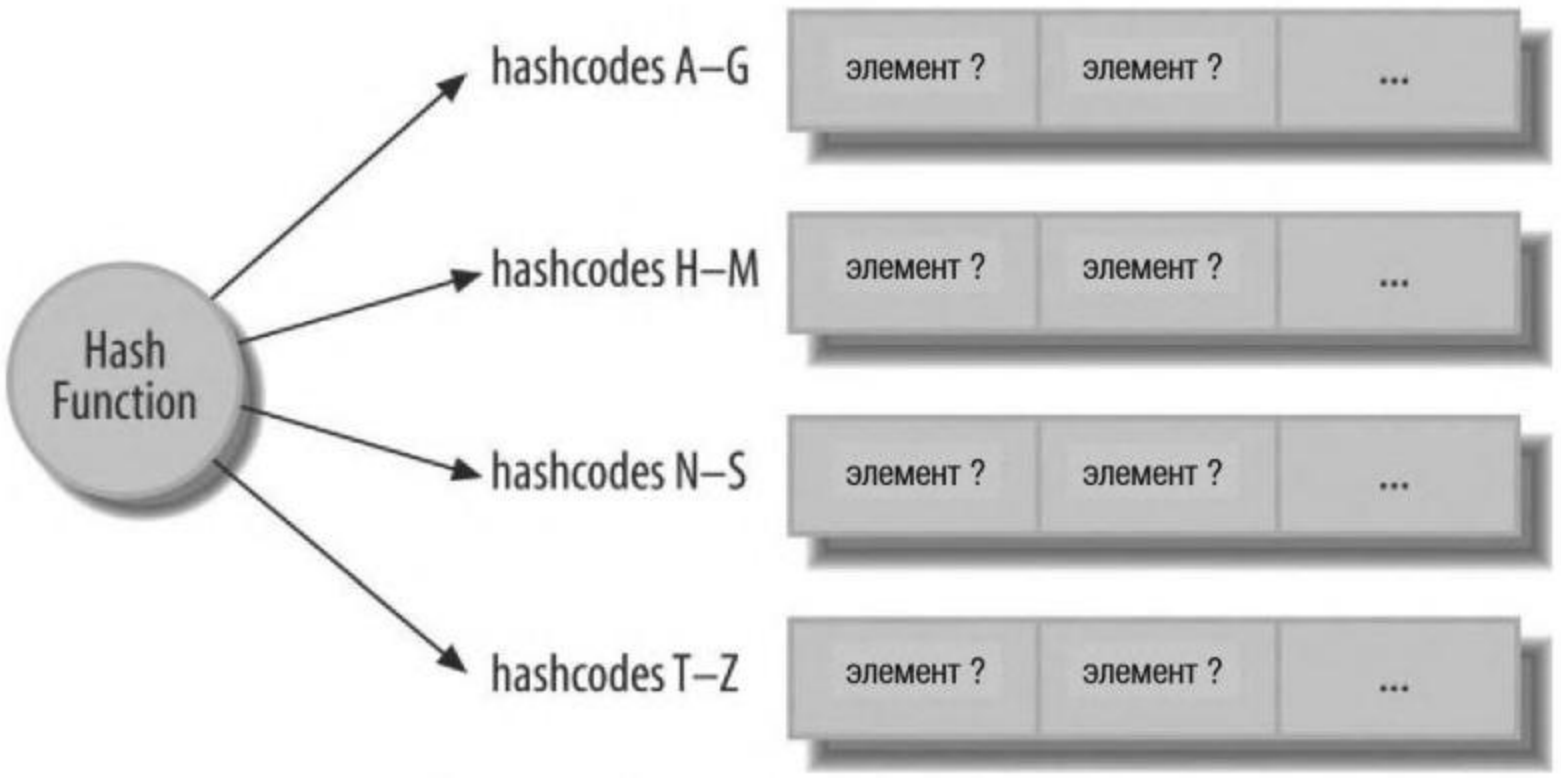

Рис. 11.5. Структура хеш-таблицы

Реализации коллекций на платформе Java

В табл. 11.7 перечислены реализации коллекций из состава технологии Java Collections Framework, сгруппированные по типам интерфейсов.

Табл. 11.7. Классы-реализации из Collections Framework

Интерфейс	Реализация
`Set`	`HashSet`
	`LinkedHashSet`
	`CopyOnWriteArraySet`
	`EnumSet`
	`CopyOnWriteArraySet`
`SortedSet`	`TreeSet`
	`ConcurrentSkipListSet`
`List`	`ArrayList`
	`LinkedList`

Интерфейс	Реализация
	Vector*
	Stack
	CopyOnWriteArrayList
Map	HashMap
	EnumMap
	LinkedHashMap
	IdentityHashMap
	Hashtable*
ConcurrentMap	ConcurrentHashMap
	ConcurrentSkipListMap
SortedMap	TreeMap
Queue / Dequeue	LinkedList
	ArrayDeque
	PriorityQueue
	DelayQueue
	SynchronousQueue
	ConcurrentLinkedQueue
	ConcurrentLinkedDequeue
BlockingQueue	ArrayBlockingQueue
	LinkedBlockingQueue
	PriorityBlockingQueue

* Классы Vector и Hashtable являются устаревшими. Вместо них рекомендуется использовать классы ArrayList и HashMap соответственно.

Классы ArrayList и LinkedList предоставляют реализации массива и связного списка на основе интерфейса List, описанного ранее. Класса ArrayList хватает для большинства задач, но если вы выполняете множество добавлений или удалений в разных частях списка, вам лучше использовать реализацию LinkedList.

Классы HashSet и HashMap предоставляют достойную реализацию интерфейсов Set и Map. В реализациях LinkedHashSet и LinkedHashMap алгоритм хеширования используется в сочетании со связным списком, который отвечает за *порядок добавления* элементов. Обратите внимание, что эти связные коллекции упорядочены, но не *отсортированы*.

Классы TreeSet и TreeMap поддерживают отсортированные коллекции при помощи древовидной структуры данных. В случае с типом

`TreeMap` сортируются значения *ключей*. Сортировка (которую мы еще обсудим позже в этой главе) выполняется при помощи так называемого *компаратора* (объекта, который сравнивает элементы).

Интерфейс `Queue` реализуется классами `LinkedList` (который к тому же реализует интерфейсы `List` и `Deque` — последний только на платформе Java 7) и `PriorityQueue`. Выставление приоритетов, свойственное классу `PriorityQueue`, обусловлено порядком сортировки, который, в свою очередь, определяется при помощи компаратора, передаваемого через конструктор. Элементы, которые в результате сравнения оказываются «наименьшими», получают самый высокий приоритет. Различные реализации интерфейса `BlockingQueue` применяют данный подход для очередей, которые поддерживают параллельные вычисления.

Наконец, класс `IdentityHashMap` является альтернативным вариантом реализации `HashMap`; он определяет связь между ключом и объектом на основе *идентичности*, а не *равенства*. Обычно любые два объекта, которые проходят проверку на равенство при помощи метода `equals()`, считаются в ассоциативном массиве одним и тем же ключом. Класс `IdentityHashMap` позволяет извлечь элемент только при помощи оригинального объекта. Подробнее о хеш-кодах и ключах мы поговорим в следующем разделе.

Также стоит упомянуть о трех специальных коллекциях, которые мы обсудим позже: классы `EnumSet` и `EnumMap` специально созданы для работы с перечислениями в языке Java; класс `WeakHashMap` использует так называемые *слабые ссылки*, чтобы влиять на процесс сборки мусора в Java-машине.

Хеш-коды и ключи

Термин *хеш* в контексте классов `Hashtable` и `HashMap` обозначает хеш ключа, который используется для установления связей в этих коллекциях. Если говорить более конкретно, то в классах `Hashtable` и `HashMap` связь между элементом и ключом определяется не уникальностью последнего, а некоей функцией, которая зависит от его содержимого. Это позволяет получать доступ к одному и тому же объекту при помощи разных, но равных (эквивалентных) ключей. Под словом «равные» мы подразумеваем, что сравнение этих двух объектов с применением метода `equals()` возвращает `true`. Элемент, сохраненный в коллекции `Hashtable` с использованием одного ключа, можно извлечь, указав дру-

гой объект, который проходит проверку на равенство при помощи метода `equals()`.

Если вы вспомните наш разговор о строках, вам будет легко понять, почему равенство так важно. Вы можете создать два объекта типа `String`, которые содержат одни и те же символы, но при этом с точки зрения языка Java являются разными объектами. В данном случае оператор `==` скажет вам, что строки отличаются, однако метод `equals()` укажет на их равенство. Поэтому сохранить элемент внутри коллекции `HashMap` можно с помощью одного из этих объектов, а извлечь — с помощью другого.

Благодаря хеш-коду объекта связь основывается на содержимом. Как мы уже упоминали в главе 7, для содержимого объекта хеш-код играет роль «отпечатков пальцев». Он используется коллекцией `HashMap` для сохранения элементов, которые можно быстро извлечь. Хеш-код представляет собой не что иное, как (целое) число, которое вычисляется на основе каких-то данных. Для одних и тех же данных число всегда остается неизменным, но сама функция хеширования написана таким образом, чтобы результаты для сочетаний разных фрагментов информации отличались как можно сильнее (и выглядели случайными). Иными словами, малейшее изменение данных должно сильно влиять на хеш-код. Вероятность того, что неидентичные наборы данных (даже если они похожи) могут иметь одинаковые хеш-коды, должна быть очень низкой.

Как уже говорилось ранее, класс `HashMap` всего лишь хранит несколько списков с элементами, однако процесс добавления этих элементов выполняется с учетом их хеш-кодов. Чтобы найти объект, класс `HashMap` проверяет его хеш-код, немедленно получая информацию о том, как добраться до нужного списка. После этого, возможно, придется перебрать несколько элементов, но их будет немного, так как сам список короткий. Чтобы определить, подходит ли ключ, для каждого объекта в найденном списке проводится следующее сравнение:

```
if ((keyHashcode == storedKeyHashcode) && key.equals(storedKey))
   return object;
```

Не существует никакого общепринятого способа генерировать хеш-коды. Есть только одно условие: ключи должны быть воспроизводимыми (в зависимости от данных) и в определенной степени случайными. Это означает, что два разных объекта могут случайно получить один и тот же хеш-код, однако это очень маловероятно (количество возможных значений типа `Integer` равно 2^{32}). Кроме того, такое совпадение не

должно вызвать никаких проблем, потому что, как вы могли заметить в предыдущем фрагменте кода, для поиска соответствующего значения класс `HashMap` всегда проверяет ключи (а также их хеш-коды) с помощью метода `equals()`. Таким образом, если два ключа не проходят проверку на равенство, они могут сосуществовать в рамках коллекции `HashMap`, несмотря на совпадение их хеш-кодов. (Другими словами, если два ключа имеют один и тот же хеш-код и метод `equals()` указывает на их равенство, они будут считаться одним ключом и с их помощью можно будет извлекать одно и то же значение).

Хеш-коды вычисляются методом `hashCode()`, оригинальная (не переопределенная) версия которого наследуется от класса `Object`. Стандартная версия метода `hashCode()` просто присваивает каждому экземпляру объекта уникальный номер, который будет использоваться в качестве хеш-кода. Так происходит во всех классах, где этот метод не переопределяется. Это хорошо сочетается со стандартной реализацией метода `equals()` из класса `Object`, который проверяет идентичность объектов с помощью обычного оператора ==; в итоге эти произвольные объекты играют роль уникальных ключей в ассоциативных массивах.

В любых классах, в которых равенство разных объектов имеет какой-то смысл, метод `equals()` нужно переопределять. Аналогично, если вы хотите, чтобы равные объекты рассматривались как один и тот же ключ, вам необходимо также переопределить метод `hashCode()`, чтобы он возвращал идентичные хеш-коды. Для этого нужно подобрать подходящий генератор случайных значений, который будет служить произвольной функцией от содержимого ваших объектов. К такой функции предъявляется только одно требование: она должна почти наверняка возвращать разные значения для объектов с разными данными, однако для идентичных объектов результаты должны совпадать.

Синхронизированные и несинхронизированные коллекции

Класс `java.util.Collections` содержит важные статические методы, которые упрощают взаимодействие с коллекциями `Set` и `Map`. Все методы в этом классе работают с интерфейсами, поэтому они не зависят от того, какую реализацию вы используете. Для начала мы рассмотрим те из них, которые предназначены для создания синхронизированных версий наших коллекций.

Большинство стандартных реализаций коллекций не синхронизировано; другими словами, их небезопасно использовать при параллельном доступе из разных потоков. Сделано это из соображений производительности.

Во многих приложениях синхронизация попросту не нужна, поэтому в API-интерфейсах коллекций она по умолчанию не поддерживается. Тем не менее вы можете создать синхронизированную версию любой коллекции, используя следующие методы из класса `Collections`:

```
public static Collection synchronizedCollection(Collection c)
public static Set synchronizedSet(Set s)
public static List synchronizedList(List list)
public static Map synchronizedMap(Map m)
public static SortedSet synchronizedSortedSet(SortedSet s)
public static SortedMap synchronizedSortedMap(SortedMap m)
```

Эти методы возвращают синхронизированные, безопасные для использования в разных потоках версии уже имеющихся коллекций, создавая для них обертку (новый объект, который реализует тот же интерфейс, делегируя вызовы исходной коллекции). В следующем фрагменте кода показан пример того, как создать объект `List`, адаптированный для работы в многопоточном режиме:

```
List list = new ArrayList();
List syncList = Collections.synchronizedList(list);
```

Методы этого списка могут безопасно вызываться из нескольких потоков. Если нужно будет дождаться завершения других потоков, эти вызовы могут блокироваться.

В отличие от более современных коллекций, интерфейсы `Hashtable` и `Vector` синхронизируются по умолчанию (и в результате могут работать немного медленнее даже в тех случаях, когда этого не требуется). Реализации коллекций типа copy-on-write (копирование при записи), о которых мы поговорим позже, не требуют отдельной синхронизации для корректной работы.

Далее также будут рассмотрены реализации `ConcurrentHashMap` и `ConcurrentLinkedQueue`, адаптированные для работы с множеством потоков; они созданы специально для эффективной поддержки параллельного доступа *без* существенного урона для собственной внутренней синхронизации.

Синхронизация итераторов

Это важный момент, и его нужно запомнить! Синхронизированные коллекции поддерживают работу с множеством потоков, однако этого нельзя сказать об итераторах, которые они возвращают.

Итератор, полученный из коллекции, нужно синхронизировать самостоятельно, следя за тем, чтобы во время перебора элементов коллекция оставалась неизменной. Это принято делать внутри самой коллекции посредством блока `synchronized`:

```
synchronized(syncList) {
   Iterator iterator = syncList.iterator();
   // здесь выполняется работа с итератором
}
```

Если пренебречь синхронизацией при переборе элементов и коллекция при этом изменится, то платформа Java попытается сгенерировать исключение `ConcurrentModificationException` (однако нет никакой гарантии того, что у нее получится это сделать).

Классы `ConcurrentHashMap` и `ConcurrentLinkedQueue`

Класс `java.util.concurrent.ConcurrentHashMap` входит в состав пакета для выполнения параллельных вычислений, представляя собой ассоциативный массив, который хорошо работает в условиях многопоточного доступа. Этот класс безопасен для работы с несколькими потоками, но достигается это не обязательно за счет блокирования потоков во время выполнения операций.

Допускается одновременное выполнение некоторых пересекающихся действий, таких как параллельное чтение. Пока выполняется доступ, класс `ConcurrentHashMap` может даже позволить ограниченное количество одновременных операций записи. Такой подход не грозит появлением исключения `ConcurrentModificationException`, но вместе с тем нельзя точно сказать, когда один поток «увидит» работу другого. При чтении всегда учитываются самые последние изменения, которые успели записаться.

То же самое можно сказать о реализации `ConcurrentLinkedQueue`. Она наделяет похожими преимуществами связную очередь, обеспечивая некоторый уровень распараллеливания для пересекающихся операций записи и чтения.

Коллекции, предназначенные только/преимущественно для чтения

С помощью класса `Collections` любую коллекцию можно сделать доступной только для чтения:

```
public static Collection unmodifiableCollection(Collection c)
public static Set unmodifiableSet(Set s)
public static List unmodifiableList(List list)
public static Map unmodifiableMap(Map m)
public static SortedSet unmodifiableSortedSet(SortedSet s)
public static SortedMap unmodifiableSortedMap(SortedMap m)
```

Создание неизменяемых версий коллекций — это удобный способ убедиться в том, что при передаче данных в другую часть вашего кода они не будут случайно модифицированы. Попытка изменить коллекцию, предназначенную только для чтения, приводит к генерированию исключения `UnsupportedOperationEx`.

Коллекции с поддержкой копирования при записи (предназначенные «преимущественно для чтения»)

Пакет `java.util.concurrent` содержит классы `CopyOnWriteArrayList` и `CopyOnWriteArraySet`, которые реализуют интерфейсы `List` и `Set`. Эти классы являются безопасными для использования в условиях многопоточности, не требуют явной синхронизации и хорошо оптимизированы для чтения. Любая операция записи приводит к автоматическому копированию всей структуры данных в блокирующем режиме. В условиях, когда коллекция предназначена преимущественно для чтения, эта реализация работает чрезвычайно быстро и совершенно не требует синхронизации.

Класс `WeakHashMap`

В главе 5 вы познакомились с концепцией слабых ссылок, которые не мешают сборщику мусора удалять связанные с ними объекты. Класс `WeakHashMap` — это реализация интерфейса `Map`, которая использует слабые ссылки как для ключей, так и для значений. Это означает, что по завершении работы с ассоциативным массивом вам не обязательно удалять его содержимое. Обычно реализация `Map` хранит ссылку на объект ключа, поддерживая его «живым» (предотвращая сборку мусора), даже если все остальные ссылки на него в вашем приложении были удале-

ны. Класс `WeakHashMap` работает иначе; он позволяет сборщику мусора удалить ключ вместе с соответствующим значением (если у него тоже не остается ссылок), как только вы удалите все остальные ссылки на объект этого ключа.

Классы `EnumSet` и `EnumMap`

Коллекции `EnumSet` и `EnumMap` специально созданы для работы с ограниченными наборами объектов, имеющих тип `enum` (перечисления рассматривались в главе 5). Перечисления в языке Java — это обычные объекты, и нет ни одной причины, почему бы вы не могли использовать их в коллекциях в качестве ключей или значений. Тем не менее классы `EnumSet` и `EnumMap` специально оптимизированы и учитывают тот факт, что наборы элементов и ключи в ассоциативном массиве соответственно могут принимать всего одно из нескольких отдельных значений. Таким образом, хранилище может обладать высокими скоростью и компактностью, используя у себя внутри битовые поля. Идея заключается в следующем: методы коллекции оптимизируются для работы с перечислениями, что позволяет заменить стандартную модель использования битовых масок и сделать операции двоичной логики необязательными. Вместо следующего кода:

```
int flags = getFlags();
if ( flags & ( Constants.ERROR | Constants.WARNING ) != 0 )
```

мы могли бы воспользоваться методами из интерфейса `Set`:

```
EnumSet flags = getFlags();
if ( flags.contains( Constants.Error) ||
   flags.contains( Constants.Warning ) )
```

Этот вариант не настолько лаконичный, но его легче понять, и выполняться он должен так же быстро.

Сортировка коллекций

Класс `Collections` включает в себя вспомогательные методы для выполнения таких рутинных операций, как сортировка. Есть две разновидности сортировки:

```
public static void sort(List list )
```

Этот метод используется для сортировки списков, чьи элементы реализуют интерфейс `java.lang.Comparable`. К счастью, этому требованию отвечают многие классы, включая `String`, `Date`, `BigInteger` и классы-обертки для простых типов (`Integer`, `Double` и т. д.).

```
public static void sort(List list, Comparator c)
```

Используйте этот метод для списков, элементы которых не реализуют интерфейс `Comparable`. Сравнение элементов берет на себя аналогичный интерфейс `java.util.Comparator`. Допустим, вы написали класс `ImaginaryNumber` и хотите отсортировать список элементов этого класса. Для этого вам нужно будет создать реализацию `Comparator`, которая знает, как сравнивать два гипотетических числа.

Отсортированные коллекции, которые мы обсуждали ранее, `SortedSet` и `SortedMap`, поддерживают информацию в определенном порядке при помощи интерфейса `Comparable`, который реализуют их элементы. Если элементы этого не делают, вы должны сами передать в конструктор своей реализации объект `Comparator`. Например:

```
Comparator myComparator = ...
SortedSet mySet = new TreeSet( myComparator );
```

На этом возможности коллекций не заканчиваются. Если вы хотите узнать больше, можете самостоятельно ознакомиться с методами `min()`, `max()`, `binarySearch()` и `reverse()`.

Захватывающий пример

Коллекции — довольно прозаическая тема, для которой сложно придумать интересный пример. В этом разделе представлено демонстрационное приложение, которое читает текстовый файл, разбивает его содержимое на слова, подсчитывает вхождения каждого слова, сортирует результат и записывает его в другой файл. Это даст вам определенное представление о том, как использовать коллекции в своих программах. В данном примере демонстрируются и другие возможности платформы Java, включая обобщенные типы, автоматическую упаковку и API-интерфейс класса `Scanner`.

```
import java.io.*;
import java.util.*;
public class WordSort
```

```
{
   public static void main(String[] args) throws IOException
   {
      if ( args.length < 2 ) {
         System.out.println("Использование: WordSort входящий_файл
исходящий_файл");
         return;
      }
      String inputfile = args[0];
      String outputfile = args[1];
      /* Создаем ассоциативный массив слов. Каждый ключ является словом,
а каждое значение - число типа Integer, обозначающее количество
появлений слова во входящем файле.
      */
      Map<String,Integer> map = new TreeMap<String,Integer>();
      Scanner scanner = new Scanner( new File(inputfile) );
      while ( scanner.hasNext() ) {
            String word = scanner.next();
            Integer count = map.get( word );
            count = ( count == null ? 1 : count +1 );
            map.put( word, count );
      }
      scanner.close();
      // получаем ключи ассоциативного массива
      List<String> keys = new ArrayList<String>( map.keySet() );
      // записываем результат в исходящий файл
      PrintWriter out = new PrintWriter( new FileWriter(outputfile) );
      for ( String key : keys )
            out.println( key + " : " + map.get(key) );
      out.close();
   }
}
```

Допустим, у вас есть входящий файл под названием *Pushkin.txt*:

```
Мне памятно другое время!
В заветных иногда мечтах
Держу я счастливое стремя...
И ножку чувствую в руках;
Опять кипит воображенье,
Опять ее прикосновенье
```

```
Зажгло в увядшем сердце кровь,
Опять тоска, опять любовь!..
```

Вы можете открыть этот файл в своем приложении с помощью следующей команды:

```
% java WordSort Pushkin count.txt
```

Исходящий файл, *count.txt*, будет выглядеть так:

```
В : 1
Держу : 1
Зажгло : 1
И : 1
Опять : 3
в : 2
воображенье, : 1
время! : 1
другое : 1
...
```

Результат зависит от регистра: слова «В» и «в» записаны отдельно. Вы можете это изменить, преобразовав все слова в нижний регистр после того, как получите их из объекта `Scanner`.

Класс Properties

Класс `java.util.Properties` представляет собой хеш-таблицу, предназначенную специально для строк. Обычно он используется для хранения текстовых конфигурационных данных. Примером этого могут служить системные свойства платформы Java, которые передаются в приложение посредством командной строки (их мы обсудим позже в этом разделе). В целом вы можете использовать таблицу `Properties` в своих приложениях для хранения произвольной конфигурационной информации в легкодоступном формате. Объект `Properties` обладает определенным изяществом и позволяет загружать и сохранять данные в формате XML или в виде обычного текста, используя потоки ввода/вывода (о них вы можете узнать в главе 12).

В таблице `Properties` можно сохранить любые строковые данные в виде пар «ключ/значение». Но чтобы сформировать из свойств логическую структуру, принято использовать иерархическое именование

с точками в качестве разделителей. (К сожалению, это всего лишь рекомендованный подход; на практике вы не можете работать с группами свойств как с иерархией, хотя это и было бы логично). Например, вы можете создать пустой объект `Properties` и добавить в него строковую пару «ключ/значение», как будто это обычный ассоциативный массив:

```
Properties props = new Properties();
props.setProperty("myApp.xsize", "52");
props.setProperty("myApp.ysize", "79");
```

Сделав это, вы можете извлечь значение с помощью метода `getProperty()`:

```
String xsize = props.getProperty( "myApp.xsize" );
```

Если указанного свойства не существует, метод `getProperty()` вернет `null`. Метод `propertyNames()` возвращает объект `Enumeration`, с помощью которого можно перечислить имена свойств:

```
for ( Enumeration e = props.propertyNames(); e.hasMoreElements(); ) {
   String name = e.nextElement();
   ...
}
```

При создании объекта `Properties` вы можете указать еще один экземпляр этого класса, который будет хранить значения по умолчанию:

```
Properties defaults = ...
Properties props = new Properties( defaults );
```

Теперь, когда метод `getProperty()` не найдет указанное свойство в текущей таблице, он начнет искать в таблице `defaults`. У метода `getProperty()` есть и альтернативная версия, которая тоже принимает значение по умолчанию; это значение возвращается вместо `null`, если свойство не было найдено ни в текущем списке, ни в таблице `defaults`:

```
String xsize = props.getProperty( "myApp.xsize", "50" );
```

Загрузка и сохранение

Вы можете сохранить таблицу `Properties` в поток `OutputStream`, используя метод `save()`. Информация о свойствах выводится в чисто текстовом формате ASCII. О вводе/выводе мы поговорим в следующей главе, но пока что вам не стоит на это отвлекаться. В продолжение пред-

ыдущего примера, мы выведем информацию о свойстве при помощи потока System.out:

```
props.save( System.out, "Application Parameters" );
```

System.out — это стандартный исходящий поток, который выводит данные в консоль или командную строку приложения.

Мы могли бы сохранить информацию в файл, указав в качестве первого аргумента для метода save() поток FileOutputStream. Второй аргумент — это строка, которая используется как заголовок. Предыдущий код выводит в поток System.out примерно такой текст:

```
#Application Parameters
#Sun Sep 22 09:24:23 EEST 2013
myApp.ysize=79
myApp.xsize=52
```

Метод load() считывает из потока InputStream ранее сохраненное содержимое объекта Properties:

```
FileInputStream fin;
...
Properties props = new Properties()
props.load( fin );
```

Метод list() может пригодиться при отладке приложения. Он выводит содержимое исходящего потока в более понятном для человека формате (хотя для метода load() он не подходит). Он обрезает длинные строки, заканчивая их многоточиями (...).

Класс Properties также содержит методы storeToXML() и loadFromXML(). Они работают точно так же, как методы save() и load(), но в качестве выходного формата используют XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>My Properties</comment>
<entry key="myApp.ysize">79</entry>
<entry key="myApp.xsize">52</entry>
</properties>
```

Подробнее об XML мы поговорим в главе 24.

Системные свойства

Класс `java.lang.System` содержит статический метод `System.getProperties()`, который предоставляет доступ к основной информации о системе. Этот метод возвращает таблицу `Properties` с системными свойствами, которые исполняют роль переменных окружения в некоторых средах программирования. В табл. 11.8 собраны системные свойства, доступные на любой платформе Java.

Табл. 11.8. Системные свойства

Системное свойство	Значение
`java.vendor`	Идентификатор изготовителя виртуальной машины JVM
`java.vendor.url`	URL-адрес изготовителя виртуальной машины JVM
`java.version`	Версия платформы Java
`java.home`	Директория, в которую установлена платформа Java
`java.class.version`	Версия библиотеки классов платформы Java
`java.class.path`	Путь к библиотеке классов платформы Java
`os.name`	Название операционной системы
`os.arch`	Архитектура операционной системы
`os.version`	Версия операционной системы
`file.separator`	Разделитель файлов (например, «/» или «\»)
`path.separator`	Разделитель пути (например, «:» или «;»)
`line.separator`	Разделитель строк (например, «\n» или «\r\n»)
`user.name`	Имя учетной записи
`user.home`	Домашняя директория пользователя

Java-апплеты и другие приложения на платформе Java, которые работают в изолированной среде, могут не иметь доступа к чтению следующих свойств: `java.home`, `java.class.path`, `user.name`, `user.home` и `user.dir`. Как вы увидите позже, эти ограничения реализуются на уровне объекта `SecurityManager`.

Ваше приложение может устанавливать системные свойства при помощи метода `System.setProperty()`.

Вы также можете создавать собственные свойства при запуске виртуальной машины Java, используя ключ `-D`:

```
% java -Dfoo=bar -Dcat=Boojum MyApp
```

Системные свойства часто используются для передачи таких параметров, как числа или цвета, поэтому платформа Java предоставляет удобные средства для получения значений таких свойств и приведения их к подходящим типам. Каждый из классов `Boolean`, `Integer`, `Long` и `Color` содержит специальный геттер, который ищет и разбирает системные свойства. Например, вызов `Integer.getInteger("foo")` ищет системное свойство под названием foo, и возвращает его в виде значения типа `Integer`.

Технология Preferences API

Технология Preferences API работает с системными и пользовательскими конфигурационными данными (настройками), обеспечивая их сохранность между запусками виртуальной машины Java. Она похожа на переносимую версию реестра операционной системы Windows и представляет собой компактную базу данных, в которой можно хранить небольшие объемы информации, доступные для всех приложений. Записи хранятся в виде пар «ключ/значение»; поддерживается несколько стандартных типов, таких как строки, числа, булевы значения и даже короткие массивы байтов. Следует подчеркнуть, что технология Preferences API не предназначена для использования в качестве полноценной базы данных и вы не можете хранить в ней большие объемы информации.

Элементы настроек хранятся в виде логической древовидной структуры. Объект `Preferences` представляет собой узел дерева с уникальным путем. Можете воспринимать конфигурационные данные как файлы внутри структуры директорий; по этой аналогии в каждом файле может находиться одна или несколько пар «ключ/значение». Чтобы сохранить или извлечь элементы, вам необходимо запросить объект `Preferences` с нужным путем. Ниже показан пример (процесс поиска узлов мы рассмотрим чуть позже):

```
Preferences prefs = Preferences.userRoot().node("oreilly/learningjava");
prefs.put("автор", " Нимейер");
prefs.putInt("издание", 4);
String author = prefs.get("автор", "неизвестен");
int edition = prefs.getInt("издание", -1);
```

Помимо значений `String` и `int`, свои методы доступа имеют и другие типы: `getLong()`, `getFloat()`, `getDouble()`, `getByteArray()` и `getBoolean()`. Каждый из этих методов принимает имя ключа и значение по умолча-

нию (на случай, если искомое значение не определено). И, конечно же, у каждого метода доступа есть свой аналог с префиксом `put`, который принимает имя ключа и значение соответствующего типа. Значения по умолчанию в методах доступа являются обязательными. Это сделано для того, чтобы приложения могли работать даже при отсутствии конфигурационной информации или в моменты, когда хранилище недоступно (о чем мы поговорим позже).

Системная и пользовательская конфигурация хранится в двух отдельных деревьях. *Системные настройки* доступны для всех пользователей системы, в рамках которой установлена платформа Java. Пользовательские настройки разделены между отдельными пользователями, каждый из которых может видеть только собственные конфигурационные данные. В предыдущем примере мы воспользовались статическим методом `userRoot()`, чтобы получить корневой узел (объект `Preferences`) для дерева пользовательских настроек. Затем при помощи метода `node()` мы запросили дочерний узел, который имеет путь *oreilly/learningjava*. Для получения системного корневого узла предусмотрен метод `systemRoot()`.

Метод `node()` принимает либо относительный, либо полный путь. В первом случае в качестве отправной точки для поиска используется путь к текущему узлу. Мы могли бы получить наш узел следующим образом:

```
Preferences prefs =
   Preferences.userRoot().node("oreilly").node("learningjava");
```

Когда методу `node()` передается полный путь, узел служит только для определения текущего дерева. Чтобы получить наш объект `Preferences`, мы могли бы передать путь */oreilly/learningjava* любому методу `node()`.

Применение технологии Preferences API для классов

Java — объектно-ориентированный язык, поэтому желание связать конфигурационные данные с классами является вполне естественным. В главе 12 вы увидите, что платформа Java предоставляет специальные средства для загрузки ресурсных файлов, связанных с файлами классов. Технология Preferences API использует этот подход для привязки узла к каждому пакету. Здесь применяется простой принцип: в качестве пути к узлу используется имя пакета, в котором точки (.) заменены на слеши (/). Все классы в пакете имеют общий узел.

Вы можете получить экземпляр узла для класса при помощи методов `Preferences.userNodeForPackage()` и `Preferences.systemNodeForPackage()`, которые принимают в качестве аргумента объект типа `Class` и возвращают узел пакета для пользовательского и системного деревьев соответственно. Например:

```
Preferences datePrefs = Preferences.systemNodeForPackage( Date.class );
Preferences myPrefs = Preferences.userNodeForPackage( MyClass.class );
Preferences morePrefs =
   Preferences.userNodeForPackage( myObject.getClass() );
```

Здесь мы использовали конструкцию `.class`, чтобы сослаться на объект `Class` для типа `Date` в системном дереве и для нашего собственного класса `MyClass` в пользовательском дереве. Тип `Date` находится в пакете `java.util`, поэтому в данном случае мы получим путь */java/util*. Вы можете получить объект `Class` для экземпляра любого класса, используя метод `getClass()`.

Хранилище конфигурационных данных

В непосредственном «создании» узлов нет никакой необходимости. Запрашивая узел, вы получаете объект `Preferences` с заданным путем внутри дерева. Данные, которые вы туда запишете, в итоге попадут в постоянное хранилище, которое еще называют «вспомогательным» (backing store). *Вспомогательное хранилище* — это механизм хранения конфигурационных данных, который зависит от конкретной реализации. Все методы с префиксом `put` отрабатывают мгновенно, не давая никакой гарантии относительно того, сохранилась ли информация на самом деле. Вы можете сохранить данные во вспомогательном хранилище принудительно, используя метод `flush()` из класса `Preferences`. Кроме того, при помощи метода `sync()` вы можете обеспечить актуальность объекта относительно всех изменений, которые были внесены во вспомогательное хранилище другими приложениями или потоками. Если по каким-то причинам данные не могут быть прочитаны или записаны, методы `flush()` и `sync()` генерируют исключение `BackingStoreException`.

Узлы не нужно создавать, но при помощи метода `nodeExists()` вы можете проверять их наличие; вы также можете удалять их вместе со всеми дочерними узлами, используя метод `removeNode()`. Чтобы удалить отдельный элемент из узла, вызовите метод `remove()` и передайте ему со-

ответствующий ключ; вы можете удалить все содержимое узла при помощи метода `clear()` (но это не то же самое, что удалить сам узел).

Некоторые нюансы вспомогательного хранилища зависят от конкретной реализации, однако технология Preferences API предоставляет простые инструменты для импорта/экспорта, которые могут считывать отдельные части дерева конфигурации, и записывать их в XML-файл (с форматом которого можно ознакомиться на странице **java.sun.com/dtd/**). Объект `Preferences` может быть записан в исходящий поток при помощи метода `exportNode()`. Метод `exportSubtree()` записывает узел вместе со всеми его дочерними ответвлениями. С другой стороны, статический метод `Preferences.importPreferences()` может прочитать XML-файл и заполнить его содержимым соответствующее дерево (системное или пользовательское); пользовательские данные всегда записываются в дерево текущего пользователя независимо от того, кто их сгенерировал.

Стоит заметить, что во время импорта запись выполняется непосредственно в дерево, поэтому вы не можете использовать этот механизм в качестве хранилища данных в формате XML (для этого предусмотрены другие API-интерфейсы). И хотя мы уже упоминали, что подробности реализации этого хранилища здесь не рассматриваются, вам, возможно, будет интересно узнать, как оно работает в своем текущем варианте. В некоторых системах платформа Java создает для каждого дерева свою иерархию директорий: *$JAVA_HOME/jre/.systemPrefs* и *$HOME/.java/.userPrefs*. Для каждого узла создается своя вложенная директория, внутри которой находится XML-файл под названием *prefs.xml*.

Уведомления об изменениях

Часто бывает нужно, чтобы во время своего выполнения программа уведомлялась о текущих изменениях в конфигурационных данных. Вы можете получать такие уведомления при помощи интерфейсов `PreferenceChangeListener` и `NodeChangeListener`; это так называемые *слушатели событий* (в главах 16–18 будет рассмотрено множество подобных интерфейсов). Позже в разделе «Интерфейс `Observer` и класс `Observable`» мы обсудим общий принцип их работы. Пока что вам достаточно знать следующее: зарегистрировав для узла объект, который реализует интерфейс `PreferenceChangeListener`, вы сможете получать уведомления о добавлении, удалении и изменении конфигурационных данных этого узла. Интерфейс NodeChange`Listener` позволяет следить за добавлением и удалением дочерних ответвлений заданного узла. Вот

пример кода, который отображает все изменения, произошедшие с содержимым нашего узла */oreilly/learningjava*:

```
Preferences prefs =
   Preferences.userRoot().node("/oreilly/learningjava");
prefs.addPreferenceChangeListener( new PreferenceChangeListener() {
   public void preferenceChange(PreferenceChangeEvent e) {
      System.out.println("Значение: " + e.getKey()
         + " поменялось на "+ e.getNewValue() );
   }
} );
```

Если вкратце, этот код отслеживает изменения в конфигурационных данных и выводит их на экран. Пока что этот пример может показаться не совсем понятным, но все должно проясниться, когда мы приступим к изучению событий (начиная с главы 16 и дальше).

API-интерфейсы для ведения журнала сообщений

Пакет `java.util.logging` предоставляет крайне гибкий и простой в использовании набор инструментов для ведения журнала системных событий и сообщений об ошибках, а также для хорошо структурированного вывода отладочной информации. Данный пакет позволяет применять фильтры к отдельным журнальным сообщениям; благодаря этому сообщения можно выводить сразу в нескольких направлениях (включая файлы и сетевые сервисы), форматируя их содержимое в соответствии с требованиями получателей.

Самое главное, большая часть подготовки к ведению журнала может быть проделана извне, во время выполнения программы; для этого можно использовать конфигурационный файл со свойствами или внешнее приложение. К примеру, на этапе выполнения кода можно установить такие свойства, чтобы журнальные сообщения отправлялись как в заданный файл формата XML, так и в системную консоль, где они будут принимать понятный для пользователя вид. Более того, для каждого направления можно указать уровень значимости и игнорировать журнальные сообщения, которые до него недотягивают. Следуя рекомендациям относительно оформления исходного кода, вы можете регулировать уровни журнальных сообщений на отдельных участках своего приложения; это позволит записывать все подробности о конкретных пакетах

и классах, не засоряя журнал лишней информацией. API-интерфейсами для ведения журнала можно управлять даже удаленно, используя интерфейсы `MBean` из состава технологии Java Management Extensions (средства сетевого управления для платформы Java).

Обзор

Любой достойный API-интерфейс для ведения журнала обязан соблюдать как минимум два основных принципа. Во-первых, он должен иметь такую производительность, чтобы разработчик мог свободно использовать журнальные сообщения, не опасаясь снижения скорости работы. По аналогии с утверждениями из языка Java (их мы обсудили в главе 4), в отключенном состоянии журнал не должен потреблять сколько-либо существенные вычислительные ресурсы. Это означает, что пока журнальные инструкции выключены, они не влияют на производительность. Во-вторых, несмотря на то, что некоторым пользователям нужны продвинутые функции и возможность конфигурирования, API-интерфейс для ведения журнала должен иметь некий упрощенный режим, который можно использовать вместо старого доброго метода `System.out.println()`, не затрачивая лишнего времени на разработку. API-интерфейс для ведения журнала, который входит в состав платформы Java, предоставляет простую модель использования, а также методы, которые делают его довольно привлекательным для разработчиков.

Регистраторы

В основе системы для ведения журнала лежит так называемый *регистратор* — экземпляр класса `java.util.logging.Logger`. Как правило, это единственный класс из данного пакета, с которым приходится иметь дело. Чтобы создать регистратор, нужно передать его имя в метод `Logger.getLogger()`. Это имя определяет место регистратора в древовидной иерархии (с единым корневым регистратором и его дочерними ответвлениями). Такая структура позволяет наследовать конфигурацию отдельным участкам дерева, благодаря чему ведение журнала можно автоматически сконфигурировать для разных частей приложения. Для любого значительного класса в пакете рекомендуется назначать отдельный экземпляр регистратора; в качестве имен принято использовать названия пакетов/классов, записанных через точку. Например:

```
package com.oreilly.learnjava;
public class Book {
   static Logger log = Logger.getLogger("com.oreilly.learnjava.Book");
```

Класс `Logger` содержит широкий выбор методов для записи сообщений в журнал; некоторые из них предоставляют очень подробную информацию, другие же ради удобства в использовании упрощают результат, сохраняя только строки. Например:

```
log.warning("Диск заполнен на 90%.");
log.info("В чат вошел новый пользователь.");
```

Методы класса `Logger` мы рассмотрим чуть позже. Названия `warning` и `info` являются примерами двух уровней ведения журнала. Всего таких уровней семь, начиная с `SEVERE` (наивысшего) и заканчивая `FINEST` (самым низким). Благодаря подобной дифференциации журнальных сообщений мы можем выбирать уровень информации, который хотим видеть во время выполнения кода. Вместо того чтобы записывать все подряд с целью дальнейшей сортировки (что негативно отразится на производительности), мы можем выбирать, какие сообщения должны быть сгенерированы. Об уровнях ведения журнала мы поговорим в следующем разделе.

Стоит также упомянуть, что в статическом поле `Logger.global` находится регистратор с именем `global`, который отлично подходит для простых приложений или прототипов. Вы можете вызывать его в тех местах вашего кода, где у вас по старой привычке используется метод `System.out.println()`:

```
Logger.global.info("Делаем что-нибудь...")
```

Обработчики

Регистраторы — это клиентский интерфейс к системе ведения журнала, но реальная работа по передаче сообщений в нужных направлениях (например, в файлы или консоль) выполняется *обработчиками*. У каждого регистратора может быть один или несколько объектов `Handler`, в число которых входят четыре предопределенных обработчика из состава API-интерфейса для ведения журнала: `ConsoleHandler`, `FileHandler`, `StreamHandler` и `SocketHandler`. Каждый из этих классов отвечает за доставку сообщений в определенное место. Класс `ConsoleHandler` используется в стандартной конфигурации для вывода сообщений в командную строку или системную консоль. `FileHandler` может направлять вывод

в файл, придерживаясь соглашения об именовании; файлы чередуются по мере заполнения. Остальные два метода направляют сообщения, соответственно, в потоки и сокеты. Существует еще один дополнительный обработчик, `MemoryHandler`, способный хранить сообщения прямо в памяти. У него есть кольцевой буфер, который может вмещать в себя определенное количество сообщений; когда этот объект публикует свои сообщения, его буфер освобождается, передавая содержимое другому назначенному обработчику.

Как мы уже говорили, регистраторам можно назначить сразу несколько обработчиков. Сообщения отправляются вверх по иерархии так, чтобы они могли попасть к обработчикам, которые принадлежат родительским регистраторам. На примере простейшей конфигурации это означает, что все сообщения в итоге попадают к обработчикам корневого регистратора. Скоро вы узнаете, как при помощи стандартных обработчиков направлять вывод в консоль, файлы и т. д.

Фильтры

Перед тем как передать сообщение вверх по иерархии, регистратор проверяет, является ли уровень этого сообщения достаточным для того, чтобы продолжать. Если проверка не была пройдена, сообщение игнорируется на этапе появления. Чтобы фильтровать сообщения не только по их уровню, но и по дополнительным критериям, вы можете наследовать класс `Filter`. Экземпляры этого класса можно применять к регистратору извне, на этапе выполнения кода; в этом они похожи на обработчики, учитывающие уровень ведения журнала, а также на объекты форматирования, которые будут рассмотрены ниже. Объект `Filter` можно подключить к конкретному экземпляру `Handler`, чтобы фильтровать записи на этапе их вывода (а не когда они только появляются).

Объекты форматирования

Внутри системы ведения журнала сообщения передаются в промежуточном формате, включая в себя всю исходную информацию. Но после попадания в обработчик они подготавливаются для вывода при помощи объекта форматирования. Пакет для ведения журнала содержит два основных класса такого рода: `SimpleFormatter` и `XMLFormatter`. Первый из них используется по умолчанию для вывода информации в консоль. Он выдает короткие, легкие для восприятия сводки журнальных сообщений. Объект `XMLFormatter` преобразует все сообщения в за-

писи формата XML. Определение этого формата можно найти по адресу **java.sun.com/dtd/**.

Уровни журнальных сообщений

В табл. 11.9 представлен список уровней журнальных сообщений, начиная с наиболее важного.

Табл. 11.9. Уровни ведения журнала

Уровень	Значение
`SEVERE`	Сбой приложения
`WARNING`	Уведомление о потенциальной проблеме
`INFO`	Сообщения, которые могут заинтересовать конечного пользователя
`CONFIG`	Подробные данные о системной конфигурации для администраторов
`FINE`, `FINER`, `FINEST`	Подробные отладочные данные для разработчика; детализация возрастает с каждым уровнем

Эти уровни можно разделить на три группы по их назначению: для пользователя, для администратора и для разработчика. Часто приложения записывают в журнал только сообщения уровня `INFO` и выше (`INFO`, `WARNING` и `SEVERE`), так как обычно они видны конечному пользователю и, следовательно, должны подходить для общего потребления. Другими словами, они должны быть написаны четко и ясно, чтобы конечный пользователь приложения мог их понять. Часто подобные сообщения выводятся в системную консоль или в виде всплывающих уведомлений.

Уровень `CONFIG` нужно использовать для относительно статической, но подробной системной информации, которая в состоянии помочь при администрировании или установке. Эта информация может касаться устанавливаемых программных модулей, характеристик системы и параметров конфигурации. Такие подробности имеют большое значение, но для конечного пользователя они не так уж и важны.

Уровни `FINE`, `FINER` и `FINEST` предназначены для разработчиков или других людей, которые знакомы с внутренним устройством приложения. Их следует использовать при отладке, в зависимости от нужной степени детализации. Кроме того, их значение вы можете определять самостоятельно. Ниже мы покажем простой пример использования всех этих уровней.

Простой пример

В следующем (нужно признать, довольно надуманном) примере мы поэкспериментируем с конфигурацией ведения журнала, используя все его уровни. Представлены сообщения всех типов, хотя их текст имеет мало смысла.

```
import java.util.logging.*;
public class LogTest {
   public static void main(String argv[])
   {
      Logger logger = Logger.getLogger("com.oreilly.LogTest");
      logger.severe("Пропало питание - используем резервный источник!");
      logger.warning("Утеряно соединение с базой данных,
переподключаемся...");
      logger.info("Запуск выполнен.");
      logger.config("Конфигурация сервера: автономная, версия JVM 1.5");
      logger.fine("Загружаем пакет для создания диаграмм.");
      logger.finer("Строим круговую диаграмму.");
      logger.finest("Начинаем сортировку методом пузырька: значение
="+42);
   }
}
```

Данный пример крайне прост. Мы получаем экземпляр регистратора, вызывая статический метод `Logger.getLogger()`, которому передается имя нашего класса. Среди разработчиков принято указывать полное имя, поэтому мы притворимся, что наш класс находится в пакете `com.oreilly`.

Теперь запустите программу LogTest. В системной консоли вы должны увидеть вывод, похожий на следующий:

```
Sep 23, 2013 3:24:36 PM LogTest main
SEVERE: Пропало питание - используем резервный источник!
Sep 23, 2013 3:24:37 PM LogTest main
WARNING: Утеряно соединение с базой данных, переподключаемся...
Sep 23, 2013 3:24:37 PM LogTest main
INFO: Запуск выполнен.
```

Мы можем наблюдать сообщения уровня `INFO`, `WARNING` и `SEVERE`; для каждого из них выводится время возникновения, а также имена класса и метода (`LogTest main`), из которых они были отправлены. Стоит заме-

тить, что сообщения более низких уровней не были показаны. Причина в том, что обычно стандартным минимальным уровнем является `INFO`; то есть в журнал попадают только те сообщения, которые имеют степень важности `INFO` или выше. Также обратите внимание на то, что вывод направлен в системную консоль, а не в какой-либо журнальный файл — это тоже поведение по умолчанию. Теперь давайте узнаем, где эта стандартная конфигурация устанавливается и как ее переопределить.

Изменение свойств журнала

Как уже было сказано в самом начале этого раздела, наверное, самой важной характеристикой API-интерфейса для ведения журнала является возможность изменения его конфигурации на этапе выполнения кода, с использованием внешних свойств или приложений. Стандартная конфигурация хранится в файле *jre/lib/logging.properties* внутри установочной директории Java. Для платформы Java этот файл свойств является стандартным (подобные файлы мы уже описывали ранее в этой главе).

Формат данного файла довольно прост. Вы можете вносить в него изменения, но делать это не обязательно. Вместо этого на этапе запуска приложения можно указать собственный файл для настройки журнала сообщений, используя системные свойства:

```
% java -Djava.util.logging.config.file=myfile.properties
```

В этой командной строке *myfile* обозначает ваш файл свойств, содержащий команды, которые мы опишем далее. Чтобы сделать использование этого файла более постоянным, вы можете указать его в виде соответствующей записи при помощи технологии Preferences API, которую мы обсуждали ранее в этой главе. Вы можете пойти еще дальше и указать настроечный файл, а также создать класс, который станет изменять конфигурацию ведения журнала, но этот способ мы рассматривать не будем.

Простейший пример файла свойств для настройки журнала сообщений выглядит примерно так:

```
# Устанавливаем стандартный уровень ведения журнала
.level = FINEST
# Направляем вывод в консоль
handlers = java.util.logging.ConsoleHandler
```

Здесь мы установили стандартный уровень ведения журнала для всего приложения, воспользовавшись свойством `.level` (название на-

чинается с точки). Мы также задали свойство `handlers`, чтобы указать экземпляр класса `ConsoleHandler`, который позволяет выводить сообщения в консоль (это поведение по умолчанию). Запустив приложение с этим файлом в качестве конфигурации для ведения журнала, вы увидите все ваши журнальные сообщения.

Но это было только для разогрева. Давайте рассмотрим более сложную конфигурацию:

```
# Устанавливаем стандартный уровень ведения журнала
.level = INFO
# Выводим в файл и консоль
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
# Настраиваем файловый вывод
java.util.logging.FileHandler.level = FINEST
java.util.logging.FileHandler.pattern = %h/Test.log
java.util.logging.FileHandler.limit = 25000
java.util.logging.FileHandler.count = 4
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
# Настраиваем консольный вывод
java.util.logging.ConsoleHandler.level = WARNING
# Уровни для отдельных классов
com.oreilly.LogTest.level = FINEST
```

В этом примере мы сконфигурировали два обработчика журнальных сообщений: первому, `ConsoleHandler`, был присвоен уровень `WARNING`, а второй, `FileHandler`, должен выводить данные в XML-файл. Обработчик `FileHandler` будет записывать сообщения уровня не ниже `FINEST` (то есть все подряд), начиная новый файл каждые 25 000 строк и не превышая лимит в четыре файла.

Имя файла хранится в свойстве `pattern`; при необходимости прямые слеши в нем автоматически заменяются на обратные (\). Специальный символ `%h` обозначает домашнюю папку пользователя. Вы также можете использовать символ `%t`, чтобы указать временную системную директорию. Если имя файла уже используется, в него после точки автоматически вставляется число (начиная с ноля). Как вариант, при помощи спецсимвола `%u` можно указать конкретное место в имени файла, куда будет вставляться уникальный номер. Аналогичным образом при смене журнальных файлов к их именам после точки добавляется число. Чтобы самому выбрать, где именно оно должно находиться, используйте идентификатор `%g`.

В нашем примере мы указали класс `XMLFormatter`. Чтобы отправлять такие же простые сообщения в консоль, мы могли бы использовать класс `SimpleFormatter`. Обработчик `ConsoleHandler` позволяет указать любой объект форматирования при помощи свойства `formatter`.

Ранее мы обещали, что вы сможете изменять уровни ведения журнала для разных частей своего кода. Чтобы это сделать, нужно установить свойства в контексте отдельно взятых регистраторов вашего приложения, используя их иерархические имена:

```
# Уровни для отдельных имен регистраторов (классов)
com.oreilly.LogTest.level = FINEST
```

Здесь мы воспользовались именем нашего тестового регистратора, чтобы установить для него уровень ведения журнала. Журнальные свойства соблюдаются в рамках иерархии, поэтому с помощью следующей строчки мы могли бы установить уровень ведения журнала для всех классов в пакете `com.oreilly`:

```
com.oreilly.level = FINEST
```

Уровни ведения журнала устанавливаются в том порядке, в котором они указаны в файле свойств, поэтому вверху должны находиться наиболее общие из них. Также стоит заметить, что уровни, применяемые к обработчикам, позволяют объектам `FileHandler` фильтровать только те сообщения, которые передаются регистраторами. Поэтому, даже если объект `FileHandler` будет иметь уровень `FINEST`, он не сможет принимать сообщения, которые проигнорировал регистратор с уровнем `SEVERE` (из регистратора в обработчик будут попадать только сообщения уровня `SEVERE`).

Класс `Logger`

В нашем примере мы использовали семь вспомогательных методов, названных в соответствии с разными уровнями ведения журнала. Существует также три группы обобщенных методов, с помощью которых можно выводить более подробную информацию. Самые главные из методов:

```
log(Level level, String msg)
log(Level level, String msg, Object param1)
log(Level level, String msg, Object params[])
log(Level level, String msg, Throwable thrown)
```

В качестве первого аргумента эти методы принимают статический идентификатор уровня ведения журнала из класса `Level`; дальше следует параметр, массив или тип исключения. В качестве идентификатора уровня могут выступать значения `Level.SEVERE`, `Level.WARNING`, `Level.INFO` и т. д.

У каждого из этих четырех методов есть аналог с именем `logp()`, который принимает названия исходных класса и метода в качестве второго и третьего аргументов соответственно. В нашем примере мы видели, как эта информация автоматически определяется платформой Java, так зачем же нам указывать ее вручную? Дело в том, что из-за динамических оптимизаций на этапе выполнения кода Java не всегда может определить точное название метода. Символ `p` в названии `logp` означает precise (точный); это говорит о том, что вы сами можете контролировать данный аспект.

Есть еще одна разновидность методов с названием `logrb()` (наверное, их лучше было бы назвать `logprb()`). Помимо названий класса и метода они принимают имя набора ресурсов. Набор ресурсов нужен для локализации сообщений (см. раздел «Наборы ресурсов» в главе 10). Если использовать более общий подход, набор ресурсов можно присвоить регистратору при его создании; для этого используется еще одна разновидность метода `getLogger`:

```
Logger.getLogger("com.oreilly.LogTest", "logMessages");
```

В любом случае имя набора ресурсов передается вместе с журнальным сообщением и поэтому может быть использовано объектом форматирования. В таком случае объект форматирования воспринимает текст сообщения как ключ, с помощью которого можно найти локализованный вариант. В локализованный текст можно передавать параметры, используя стандартный формат записи сообщений; для этого существует специальная разновидность метода `log()`, которая принимает массив аргументов.

Наконец, есть вспомогательные методы `entering()`, `exiting()` и `throwing()`, с помощью которых разработчики могут записывать в журнал подробную отладочную информацию.

Производительность

В начале этого раздела мы говорили, что для API-интерфейсов, которые занимаются ведением журнала, большое значение имеет произво-

дительность. Это подтверждается тем фактом, что журнальные сообщения фильтруются в момент возникновения при помощи разных уровней, что позволяет отсеивать их на раннем этапе. За счет этого экономятся существенные ресурсы, которые могли бы пойти на их обработку. Тем не менее это не избавляет нас от определенной подготовительной работы, которую нужно выполнить до того, как начнется запись сообщений в журнал. В частности, при передаче аргументов в журнальные методы часто приходится формировать подробные сообщения или преобразовывать объекты в строки. Такие операции бывают затратными. Чтобы избежать создания лишних строк, ресурсоемкий код нужно размещать внутри проверочного блока; в качестве условия необходимости выполнения операции используйте метод `isLoggable()` из класса `Logger`:

```
if ( log.isLoggable( Level.CONFIG ) ) {
   log.config("Конфигурация: "+ loadExpensiveConfigInfo() );
}
```

Интерфейс Observer и класс Observable

Являясь относительно небольшими инструментами, интерфейс `java.util.Observer` и класс `java.util.Observable` реализуют для платформы Java фундаментальный шаблон проектирования. Оба они входят в состав фреймворка MVC (Model-View-Controller — модель-представление-контроллер) — абстракции, которая позволяет множеству клиентских объектов (представленных интерфейсом `Observer`) получать уведомления об изменениях, происходящих в определенном объекте или ресурсе (класс `Observable`). Мы неоднократно будем сталкиваться с данным подходом при работе с механизмом передачи событий, который рассматривается в главах 16–19. Эти классы не так уж часто используются напрямую, но чтобы понять принцип их работы, с ними стоит познакомиться поближе.

Объект `Observable` (он же *издатель*) содержит метод, посредством вызова которого регистрируется экземпляр `Observer` (он же *подписчик*). Когда происходят изменения, издатель отправляет уведомление, вызывая определенный метод у каждого подписчика. Подписчики реализуют интерфейс `Observer`, который предусматривает вызов метода `update()` при получении уведомления.

В следующем примере создается издатель `MessageBoard`, который хранит сообщение типа `String`. Этот объект наследует класс `Observable`

вместе с его механизмами регистрации (`addObserver()`) и оповещения (`notifyObservers()`) подписчиков. Чтобы подключиться к издателю, мы создаем объект `Student`; являясь реализацией интерфейса `Observer`, он может следить за изменениями сообщения:

```
//файл: MessageBoard.java
import java.util.*;
public class MessageBoard extends Observable {
   private String message;
   public String getMessage() {
      return message;
   }
   public void changeMessage( String message ) {
      this.message = message;
         setChanged();
         notifyObservers( message );
      }
   public static void main( String [] args ) {
      MessageBoard board = new MessageBoard();
      Student bob = new Student();
      Student joe = new Student();
      board.addObserver( bob );
      board.addObserver( joe );
      board.changeMessage("Больше домашних заданий!");
   }
} // конец класса MessageBoard
class Student implements Observer {
   public void update(Observable o, Object arg) {
      System.out.println( "Изменения на доске объявлений: " + arg );
   }
}
```

Наш объект `MessageBoard` является наследником класса `Observable`, поэтому он содержит метод `addObserver()`, с помощью которого регистрируется каждый экземпляр `Student`. Чтобы получать уведомления, объекты `Student` используют метод `update()`. Строка с сообщением изменяется посредством метода `changeMessage()` из объекта `MessageBoard`; когда это происходит, издатель вызывает методы `setChanged()` и `notifyObservers()`, чтобы оповестить своих подписчиков. В качестве аргумента метод `notifyObservers()` может принимать значение типа `Object`, которое сигнализирует об изменении. В нашем случае этим зна-

чением является объект `String`, содержащий новое сообщение; он выступает в роли второго аргумента для метода `update()`. Первым аргументом является сам издатель, то есть объект `MessageBoard`.

Метод `main()` из класса `MessageBoard` создает нового издателя, регистрирует для него два объекта `Student` и затем меняет сообщение. Когда вы запустите код, каждый объект `Student` должен вывести сообщение, сигнализируя о том, что уведомление получено.

Вы могли бы самостоятельно реализовать отношение «издатель/подписчик», воспользовавшись коллекцией `List` для хранения списка объектов `Observer`. В главе 16 и далее вы увидите, что этот шаблон проектирования является основой событийной модели в библиотеках AWT и Swing, которые используют строго типизированные варианты издателей и подписчиков (в рамках этой модели они будут называться *событиями* и *слушателями*). Но сейчас мы заканчиваем с основными инструментами и переходим к еще одной фундаментальной теме — вводу/выводу.

Глава 12
СРЕДСТВА ВВОДА/ВЫВОДА

В этой главе мы продолжаем изучать API-интерфейсы платформы Java. На очереди классы из пакетов `java.io` и `java.nio`, предоставляющие широкий набор инструментов для основных операций ввода/вывода, а также технологию, на базе которой в Java построена вся работа с файлами и сетью.

Иерархия классов этих пакетов представлена на рис. 12.1.

Сначала мы рассмотрим классы потоков из пакета `java.io`, которые являются наследниками таких фундаментальных компонентов, как `InputStream`, `OutputStream`, `Reader` и `Writer`. Затем мы изучим класс `File` и узнаем, какие средства для чтения и записи файлов предлагает пакет `java.io`. Мы затронем тему сжатия и сериализации данных, а также познакомимся с пакетом `java.nio`. Технология NIO (new I/O — новый ввод/вывод) появилась вместе с платформой Java 1.4, принеся с собой новые важные возможности для построения высокопроизводительных сервисов. В целом же она предоставляет обновленные и улучшенные API-интерфейсы, которыми можно заменить часть функциональности из пакета `java.io`.

Потоки ввода/вывода

Наиболее фундаментальные средства ввода/вывода на платформе Java основаны на *потоках ввода/вывода*. Поток представляет процесс перемещения данных между двумя компонентами (по крайней мере, в теории), один из которых *записывает*, а другой *читает*. Используя пакет `java.io` для выполнения ввода/вывода в терминале, чтения или записи файлов, установления связи посредством сокетов, вы задействуете различные виды потоков. Позже в этой главе мы рассмотрим пакет NIO, в котором используется похожая концепция под названием *каналы*. Главная разница между потоками и каналами заключается в том, что первые ориентированы на байты, а вторые — на «буферы», которые содержат разные типы данных; в целом они выполняют примерно одну

и ту же работу. Давайте начнем с перечисления видов потоков, которые нам доступны:

`InputStream`, `OutputStream`

Абстрактные классы, которые определяют базовые возможности чтения и записи неструктурированных последовательностей байтов. Они лежат в основе всех остальных байтовых потоков на платформе Java.

`Reader`, `Writer`

Абстрактные классы, которые определяют базовые возможности чтения и записи последовательностей символов с поддержкой Unicode. На их основе построены все другие символьные потоки на платформе Java.

`InputStreamReader`, `OutputStreamWriter`

Классы, которые являются связующим звеном между байтовыми и символьными потоками. Для преобразования они используют специальную кодировку символов. (Запомните: в формате Unicode символ не эквивалентен байту!)

`DataInputStream`, `DataOutputStream`

Специальные потоковые фильтры, которые позволяют считывать и записывать многобайтовые типы данных, такие как простые числовые и строковые объекты, используя при этом универсальный формат.

`ObjectInputStream`, `ObjectOutputStream`

Специальные потоковые фильтры, способные записывать и восстанавливать целые группы сериализованных объектов.

`BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter`

Специальные потоковые фильтры, которые в целях повышения эффективности используют буферизацию. В реальных задачах ввода/вывода такой подход применяется почти всегда.

`PrintStream`, `PrintWriter`

Специальные потоки, которые упрощают вывод текста.

`PipedInputStream`, `PipedOutputStream`, `PipedReader`, `PipedWriter`

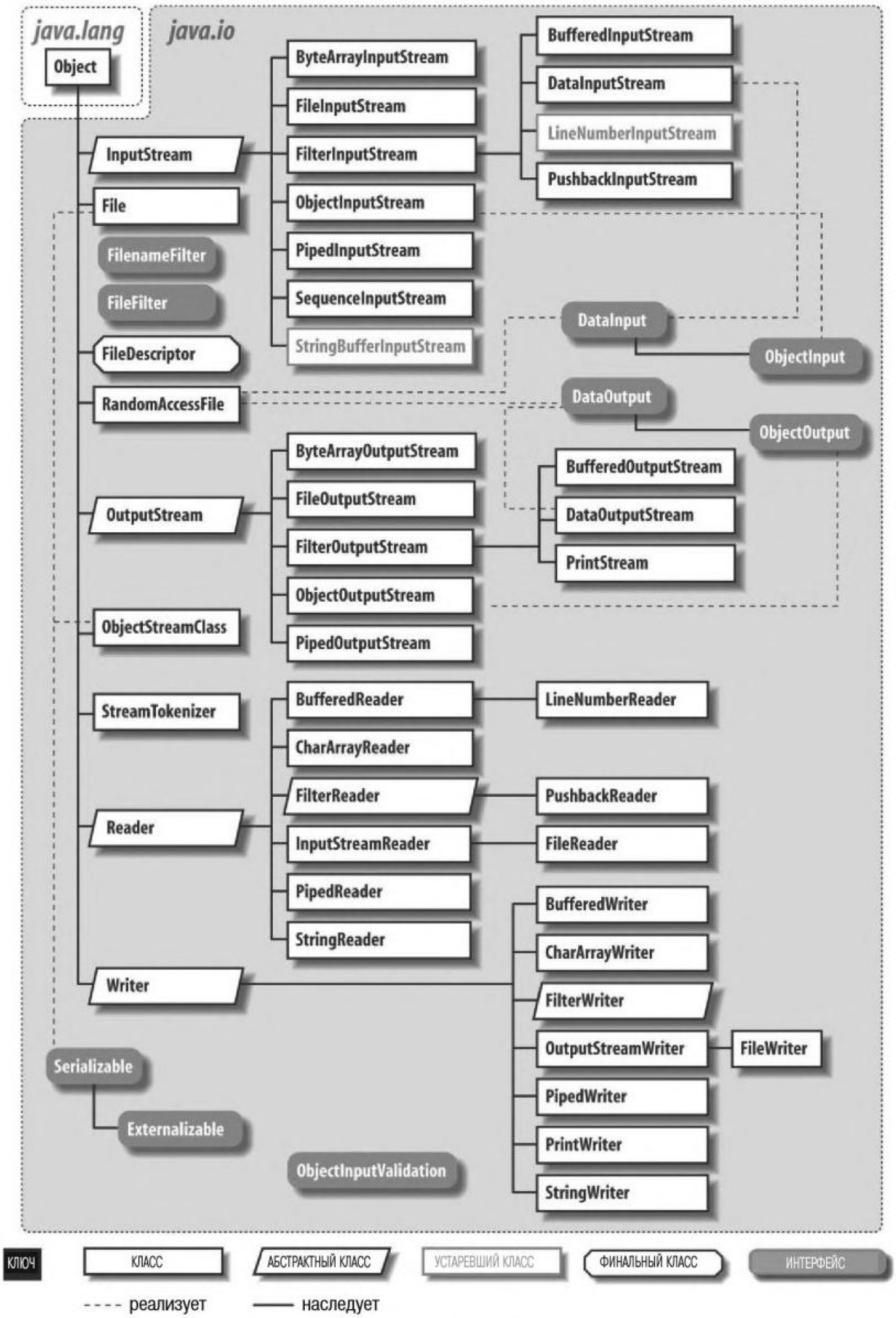

Рис. 12.1. Пакет `java.io`

«Замкнутые» потоки, которые могут использоваться парами для перемещения данных внутри приложения. Данные, записанные в по-

токи PipedOutputStream и PipedWriter, считываются из объектов PipedInputStream и PipedReader соответственно.

FileInputStream, FileOutputStream, FileReader, FileWriter

Реализации абстрактных классов InputStream, OutputStream, Reader и Writer, которые занимаются чтением и записью файлов в рамках локальной файловой системы.

На платформе Java потоки всегда направлены в одну сторону. Классы ввода и вывода из пакета java.io представляют оба конца простого потока, как показано на рис. 12.2. Для двунаправленного взаимодействия нужно использовать по одному потоку каждого типа.

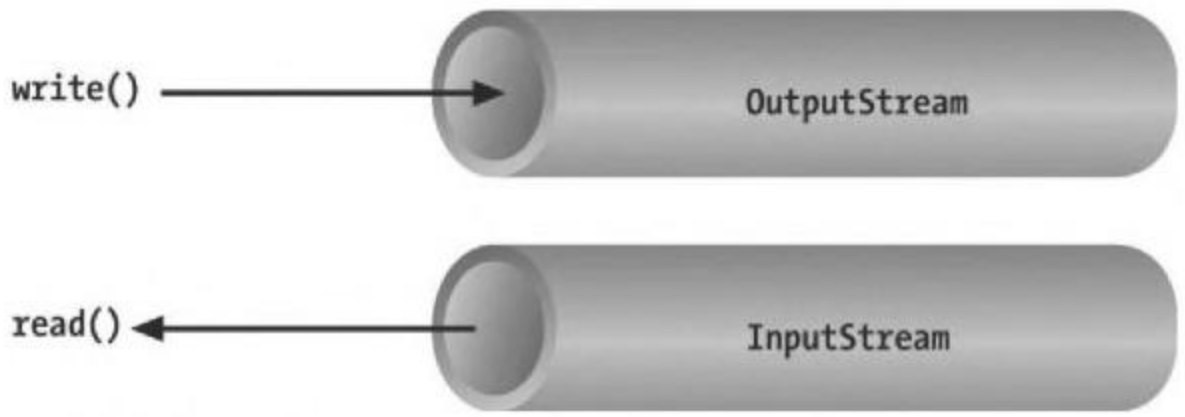

Рис. 12.2. Основная функциональность входящих и исходящих потоков

InputStream и OutputStream — это *абстрактные* классы, которые описывают интерфейс самого низкого уровня для всех байтовых потоков. Они содержат методы для чтения и записи неструктурированных наборов данных на уровне байтов. Ввиду абстрактности этих классов вы не можете создавать на их основе обобщенные потоки ввода/вывода. В составе платформы Java имеется несколько производных классов, которые предназначены для чтения и записи файлов, а также обмена информацией с помощью сокетов. Различные виды байтовых потоков могут быть взаимозаменяемыми, так как все они наследуют структуру классов InputStream или OutputStream. Метод, аргументом которого является объект InputStream, может принимать любую производную типа InputStream. Узкоспециализированные виды потоков могут наслаиваться вокруг базовых структур ввода/вывода, дополняя их такими функциями, как буферизация, фильтрация или обработка высокоуровневых типов данных.

Классы Reader и Writer очень похожи на InputStream и OutputStream, только они имеют дело с символами, а не байтами. Это настоящие символьные потоки, поэтому они умеют корректно работать с кодировкой

Unicode, что не всегда можно сказать о потоках байтов. На практике часто требуется связывать между собой два физических устройства, которые поддерживают потоки разных типов (это могут быть жесткий диск и сетевой адаптер). Специально для таких случаев предусмотрены классы `InputStreamReader` и `OutputStreamWriter`; они используют особую *кодировку*, чтобы преобразовывать символьные потоки в байтовые (и наоборот).

В этом разделе описываются все достойные внимания типы потоков, за исключением `FileInputStream`, `FileOutputStream`, `FileReader` и `FileWriter`. Обсуждение файловых потоков мы прибережем для следующего раздела. Там же мы рассмотрим проблемы, связанные с доступом к файловой системе (ФС) в контексте платформы Java.

Основы ввода/вывода

Типичным примером использования объекта `InputStream` является *стандартный ввод* Java-приложения. Как библиотека `stdin` в языке C или объект `cin` в C++, он служит источником ввода для консольных программ (без графического пользовательского интерфейса). Это входящий поток, принимающий данные из системной среды — обычно это окно терминала или, как вариант, вывод другой команды. Класс `java.lang.System` является основным хранилищем ресурсов, связанных с системой; при помощи статической переменной `System.in` он предоставляет ссылку на стандартный входящий поток. Для *стандартного исходящего потока* (`stdout`) и *стандартного потока ошибок* (`stderr`) предусмотрены переменные `out` и `err` соответственно.* Ниже показана их связь с классами потоков:

```
InputStream stdin = System.in;
OutputStream stdout = System.out;
OutputStream stderr = System.err;
```

Данный фрагмент кода скрывает тот факт, что переменные `System.out` и `System.err` — это не просто объекты `OutputStream`; они имеют более специализированный и практичный тип `PrintStream`. Причину этого мы объясним позже, ну а пока что вы можете ссылаться на переменные `out`

* Стандартный поток `stderr` часто отводится для текстовых сообщений об ошибках, которые должны выводиться пользователю в консольной программе. Он отделен от стандартного вывода, который часто перенаправляется в файл или другое приложение, из-за чего пользователь может его не увидеть.

и err как на объекты OutputStream, потому что они все равно являются производными от этого типа.

С помощью метода read() из класса InputStream мы можем считывать данные из стандартного ввода побайтно. Внимательно взглянув на API-интерфейс этого класса, вы увидите, что метод read() является абстрактным. Переменная System.in — это конкретная реализация потока InputStream, которая предоставляет реальную версию метода read():

```
try {
   int val = System.in.read();
} catch ( IOException e ) {
   ...
}
```

И хотя мы сказали, что метод read() считывает байтовые значения, типом возвращаемого результата в этом примере является int, а не byte. Дело в том, что на платформе Java для выделения конца потока данный метод использует специальное значение, следуя стандарту из языка C. Байты возвращаются в виде беззнаковых целых чисел в диапазоне от 0 до 255; число –1 используется специально для того, чтобы сигнализировать об окончании потока. При использовании простой версии метода read() вам придется проверять это условие. При необходимости вы можете привести значение к типу byte. В следующем примере на экран выводится содержимое каждого байта, прочитанного из входящего потока:

```
try {
   int val;
   while( (val=System.in.read()) != -1 )
      System.out.println((byte)val);
} catch ( IOException e ) { ... }
```

По этим примерам видно, что в случае ошибки, возникшей во время чтения исходных данных из потока, метод read() генерирует исключение IOException. У этого исключения есть разные производные, которые могут сигнализировать об ошибках в таких источниках, как файлы или сетевые соединения. Кроме того, высокоуровневые потоки, которые считывают более сложные типы данных (по сравнению с однобайтными значениями), могут генерировать исключение EOFException («конец файла»), которое указывает на неожиданное или преждевременное завершение потока.

Перегруженная версия метода `read()` наполняет байтовый массив как можно большим объемом данных, который тот способен вместить, и возвращает количество прочитанных байтов:

```
byte [] buff = new byte [1024];
int got = System.in.read( buff );
```

Теоретически при помощи метода `available()` в любой момент времени можно узнать количество байтов, доступных в потоке `InputStream`. Обладая этой информацией, мы могли бы создать массив именно такого размера, который нам нужен:

```
int waiting = System.in.available();
if ( waiting > 0 ) {
   byte [] data = new byte [ waiting ];
   System.in.read( data );
   ...
}
```

Однако надежность такого подхода зависит от того, способна ли реализация исходного потока определить объем данных, доступных для извлечения. В целом этот метод подходит для работы с файлами, но не следует считать его универсальным средством для любых типов потоков.

Методы `read()` блокируются, пока не будут прочитаны какие-либо данные (хотя бы один байт). В целом вы должны проверять возвращаемое значение, чтобы определить, сколько данных вы получили и нужно ли вам продолжать чтение (неблокирующий ввод/вывод мы рассмотрим позже в этой главе). Метод `skip()` из объекта `InputStream` дает возможность пропускать какое-то количество байтов. Для некоторых реализаций потоков это может оказаться эффективней обычного чтения.

Метод `close()` закрывается поток и освобождает все связанные с ним системные ресурсы. Большинство видов потоков необходимо закрывать по завершении работы с ними — это важно с точки зрения производительности. В некоторых случаях потоки закрываются автоматически в результате работы сборщика мусора, но на это лучше не рассчитывать. С выходом седьмой версии язык Java обзавелся конструкцией *try-with-resources*; она специально предназначена для упрощения автоматического закрытия потоков и других сущностей, которые в этом нуждаются. Позже в этой главе мы рассмотрим несколько примеров ее использования. Все типы потоков, каналов и связанные с ними вспомогательные

классы, которые могут быть закрыты, идентифицируются с помощью интерфейса-маркера `java.io.Closeable`.

И напоследок мы должны упомянуть о дополнении к стандартным потокам `System.in` и `System.out`, которое платформа Java предоставляет при помощи метода `System.console()`. Речь идет об API-интерфейсе `java.io.Console`, который позволяет считывать пароли, не отображая их на экране.

Потоки символов

В ранних версиях платформы Java некоторые разновидности реализаций `InputStream` и `OutputStream` содержали методы для чтения и записи строк, но большинство из этих методов наивно полагало, что 16-битный символ в формате Unicode равняется одному байту (8 битам) данных, взятых из потока. Это срабатывало только для кодировки Latin-1 (ISO 8859-1), но для множества других кодировок, которые используются в различных языках, такой подход не годился. В главе 10 мы узнали, что у класса `java.lang.String` есть конструктор и связанный с ним метод `getBytes()` которые помимо массива байтов принимают кодировку символов. Теоретически, с их помощью мы могли бы преобразовывать массивы байтов в символы формата Unicode; это позволило бы нам работать с байтовыми потоками, которые состоят из символов, представленных в любой кодировке. К счастью, нам не нужно этим заниматься, так как на платформе Java подобную работу выполняют сами потоки.

Классы `Reader` и `Writer` из пакета `java.io` были созданы специально для того, чтобы работать с потоками символов. Они позволяют воспринимать данные в виде символов и строк, не задумываясь о преобразовании байтов в какую-то определенную кодировку. Как мы вскоре убедимся, у классов `Reader` и `Writer` есть непосредственные реализации, которые могут, к примеру, читать и записывать файлы. Но в целом для наведения мостов между мирами символьных и байтовых потоков существует два специальных класса — `InputStreamReader` и `OutputStreamWriter`. Это такие обертки для компонентов `Reader` и `Writer`, которые позволяют превратить любой байтовый поток в символьный. Для преобразования значений (которые могут занимать несколько байт) в символы формата Unicode используется определенная кодировка. Название кодировки можно указать в конструкторах классов `InputStreamReader` и `OutputStreamWriter`. Для удобства конструкторы по умолчанию применяют стандартную системную кодировку.

Давайте для примера сконвертируем обычную строку из стандартного ввода в целое число. Мы исходим из того, что байты, полученные из потока System.in, используют стандартную системную кодировку:

```
try {
   InputStream in = System.in;
   InputStreamReader charsIn = new InputStreamReader( in );
   BufferedReader bufferedCharsIn = new BufferedReader( inReader );
   String line = bufferedCharsIn.readLine();
   int i = NumberFormat.getInstance().parse( line ).intValue();
} catch ( IOException e ) {
} catch ( ParseException pe ) { }
```

Сначала мы создаем обертку InputStreamReader вокруг потока System.in. Этим мы преобразовываем байты, поступающие из входящего потока, в обычные символы, имеющие кодировку по умолчанию. Затем мы заключаем экземпляр InputStreamReader внутрь объекта BufferedReader. BufferedReader предоставляет метод readLine(), с помощью которого строчку текста (заканчивающуюся специальной комбинацией символов, которая зависит от конкретной платформы) можно превратить в строку типа String. После этого, используя методику, описанную в главе 10, мы превращаем строку в целое число.

Обратите внимание: мы взяли входящий поток, ориентированный на передачу байтов (System.in), и безопасно преобразовали его в объект Reader, предназначенный для чтения символов. Мы могли бы указать какую-то другую кодировку, передав ее в конструктор класса InputStreamReader следующим образом:

```
InputStreamReader reader = new InputStreamReader( System.in, "UTF-8" );
```

Для формирования каждого символа, который мы в итоге получаем, объекту InputStreamReader нужно прочитать один или несколько байтов и преобразовать их в формат Unicode.

В главе 13 мы будем использовать объекты InputStreamReader и InputStreamWriter при создании простого веб-сервера; там они помогут менять кодировку символов в соответствии со значением, указанным в протоколе HTTP. К теме кодировок мы еще вернемся во время обсуждения API-интерфейса java.nio.charset, который позволяет запрашивать буферы с символами и байтами, используя конкретные экземпляры кодировщиков и декодеров. Помимо названия кодировки конструкторы классов InputStreamReader и OutputStreamWriter могут принимать объект Charset (кодек — кодировщик и декодер в одном флаконе).

Обертки для потоков

А что если нам недостаточно просто читать и записывать последовательности байтов или символов? На этот случай существует так называемый фильтрующий поток; это разновидность объектов `InputStream`, `OutputStream`, `Reader` или `Writer`, которая заключает внутрь себя существующий поток, добавляя к нему новые возможности. Поток передается в качестве аргумента для конструктора; фильтр делегирует ему поступающие вызовы, выполняя при этом какие-то дополнительные действия. Например, мы можем создать объект `BufferedInputStream`, который будет служить оберткой для стандартного системного ввода:

```
InputStream bufferedIn = new BufferedInputStream( System.in );
```

`BufferedInputStream` — это такой фильтрующий поток, который заранее считывает определенный объем данных и размещает их в буфере (мы еще к этому вернемся). Объект `BufferedInputStream` формирует дополнительную функциональную прослойку вокруг исходного потока. На рис. 12.3 показано, как это работает; в качестве примера взят поток `DataInputStream`, который способен считывать более высокоуровневые структуры данных, такие как строки и простые типы языка Java.

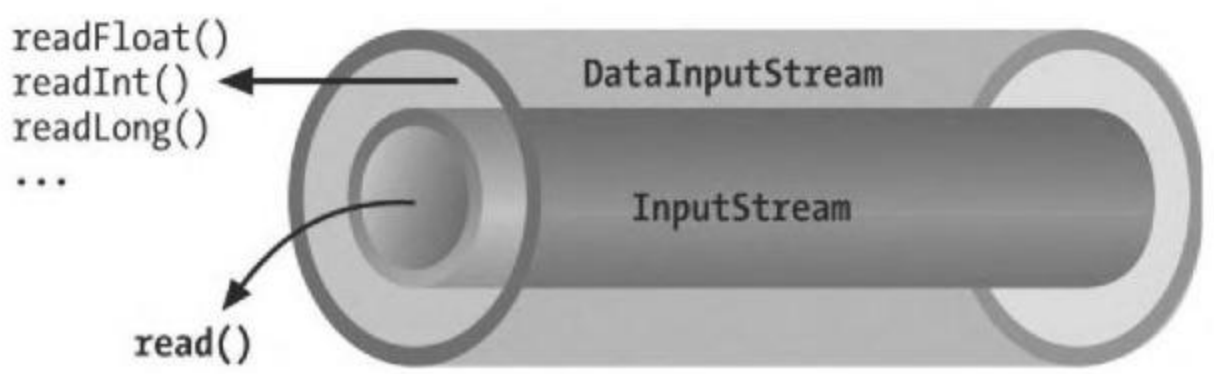

Рис. 12.3. Наслоение потоков

Как вы могли видеть в предыдущем фрагменте кода, объект `BufferedInputStream` имеет тип `InputStream`. Фильтры как таковые являются производными от базовых потоков, поэтому они могут служить аргументами для конструкторов других фильтров. Благодаря этому мы

можем «наслаивать» их друг на друга, получая сочетания разных возможностей. К примеру, чтобы считывать особые типы данных при помощи буфера, мы могли бы заключить наш поток `System.in` внутрь фильтра `BufferedInputStream`, а потом все это передать в объект `DataInputStream`.

Java предоставляет базовые классы для создания новых видов фильтрующих потоков: `FilterInputStream`, `FilterOutputStream`, `FilterReader` и `FilterWriter`. Все они содержат «холостой» механизм фильтрации; сам по себе он ничего не делает, а только делегирует вызовы всех своих методов в исходный поток. Для получения настоящих фильтрующих потоков эти классы нужно наследовать, переопределяя различные методы с целью добавления нужных функций. Позже в этой главе мы создадим свой собственный фильтрующий поток.

Потоки данных

Классы `DataInputStream` и `DataOutputStream` реализуют, соответственно, интерфейсы `DataInput` и `DataOutput`; это фильтрующие потоки, которые позволяют читать и записывать объекты `String` и любые простые типы данных, доступные в языке Java (включая булевы значения) и состоящие из более чем одного байта. Класс `DataOutputStream` кодирует данные в аппаратно-независимый формат, и затем записывает их в свой исходный поток байтов. Класс `DataInputStream` выполняет обратную процедуру.

Вы можете создать поток `DataInputStream` из объекта `InputStream`, после чего использовать метод `readDouble()` для чтения простых типов данных:

```
DataInputStream dis = new DataInputStream( System.in );
double d = dis.readDouble();
```

В этом примере стандартный входящий поток помещается внутрь объекта `DataInputStream` и с его помощью считывает значение типа `double`. Метод `readDouble()` читает байты из потока, создавая на их основе числа с плавающей запятой. Методы объекта `DataInputStream` ожидают, что числовые данные будут соблюдать *сетевой порядок байтов* — стандарт, согласно которому «старшие» байты передаются первыми (этот порядок еще называют big-endian — но об этом мы поговорим позже).

Класс `DataOutputStream` предоставляет методы для записи, которые согласуются с операциями чтения из класса `DataInputStream`. Например,

метод `writeInt()` записывает в оригинальный исходящий поток целые числа в двоичном формате.

Методы `readUTF()` и `writeUTF()` из классов `DataInputStream` и `DataOutputStream` читают и записывают объекты `String` формата Unicode, используя «промежуточную» кодировку. UTF-8 — это широко используемая разновидность кодировки Unicode, совместимая с форматом ASCII. В отличие от некоторых других кодировок, она поддерживает все символы, входящие в набор Unicode. Вы можете использовать ее совместно с потоками `Reader` и `Writer`, если укажете «UTF-8» в качестве названия кодировки.

Буферизированные потоки

Классы `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader` и `BufferedWriter` добавляют к потоку буфер данных определенного размера. Буфер может улучшить производительность, уменьшая количество реальных операций чтения и записи, которые связаны с вызовами методов `read()` и `write()`. Вы можете буферизировать нужный вам входящий или исходящий поток, указав размер буфера. (Поверх полученного результата можно разместить еще один поток, который будет пользоваться преимуществами буферизации.) Ниже показан пример входящего буферизированного потока под названием `bis`:

```
BufferedInputStream bis = new BufferedInputStream(myInputStream, 32768);
...
bis.read();
```

В этом примере мы задали буфер размером 32 Кб. Мы могли этого не делать, и тогда бы конструктор сам выбрал подходящий размер. (На сегодняшний день значение по умолчанию равно 8 Кб.) При первом вызове метода `read()` переменная `bis` пытается заполнить данными (если таковые имеются) все 32 Кб нашего буфера, откуда они будут извлекаться при последующих вызовах. Сам буфер будет автоматически наполняться по мере необходимости.

Объект `BufferedOutputStream` работает схожим образом. Чтобы поместить данные в буфер, нужно вызвать метод `write()`; сама запись будет выполнена только тогда, когда буфер заполнится. Хотя вы можете в любой момент записать содержимое потока `BufferedOutputStream`, вызвав метод `flush()` (который на самом деле принадлежит классу `OutputStream`). Благодаря этому вы можете быть уверены, что все дан-

ные со всех исходных и фильтрующих потоков будут отправлены по назначению (и для этого вам не нужно будет ждать ответа).

Некоторые входящие потоки, такие как `BufferedInputStream`, способны маркировать данные, чтобы позже начинать чтение с определенной позиции. Метод `mark()` устанавливает для потока точку возврата. Он принимает целое число, представляющее собой количество байтов, которые могут быть прочитаны до завершения потока, и «забывает» о предыдущей отметке. Метод `reset()` возвращает поток к отмеченной позиции; все данные, которые были прочитаны после вызова метода `mark()`, опять попадут в поток.

Эта функциональность может пригодиться при разборе содержимого потока. Некоторые структуры не поддаются распознаванию; в таких ситуациях вы можете сгенерировать ошибку и откатиться до момента, который предшествовал началу разбора структуры:

```
BufferedInputStream input;
...
try {
   input.mark( MAX_DATA_STRUCTURE_SIZE );
   return( parseDataStructure( input ) );
}
catch ( ParseException e ) {
   input.reset();
   ...
}
```

Классы `BufferedReader` и `BufferedWriter` работают так же, как их «байтовые» аналоги, только вместо байтов они оперируют символами.

Потоки PrintWriter и PrintStream

Класс `java.io.PrintWriter` — еще одна полезная обертка вокруг потоков. Он предоставляет набор перегруженных методов `print()`, которые превращают переданные им аргументы в строки и переправляют их в поток. У каждого из этих методов есть своя вспомогательная версия `println()`, которая добавляет к результату идентификатор перевода строки. Для вывода форматированного текста существуют вызовы `printf()` и `format()` (отличающиеся только названием), которые позволяют записывать в поток текст, отформатированный в стиле классической функции `printf` из языка C.

PrintWriter отличается от других символьных потоков: он может служить оберткой как для класса OutputStream, так и для любой производной класса Writer. Он больше похож на современную версию байтового потока PrintStream. Потоки System.out и System.err являются объектами типа PrintStream; вы уже могли видеть их в разных частях этой книги:

```
System.out.print("Здравствуй, мир...\n");
System.out.println("Здравствуй, мир...");
System.out.printf("Ответ: %d", 17 );
System.out.println( 3.14 );
```

Классы Reader и Writer отсутствовали в ранних версиях платформы Java, а вместо них использовался поток PrintStream, который конвертировал байты в символы на основе определенной кодировки. В разработке любых новых приложений нужно использовать класс PrintWriter.

При создании объекта PrintWriter вы можете передать в его конструктор дополнительную переменную типа Boolean, которая определяет, нужно ли автоматически «сбрасывать» содержимое потока. Если присвоить этой переменной значение true, объект PrintWriter будет вызывать метод flush() каждый раз, когда исходные потоки OutputStream или Writer шлют идентификатор перевода строки:

```
PrintWriter pw = new PrintWriter( myOutputStream, true /*автосброс*/ );
// Поток автоматически сбрасывается с каждым пеерводом строки.
pw.println("Привет!");
```

Использование данного подхода в сочетании с буферизацией исходящего потока похоже на построчный вывод информации в терминале.

Еще одно значительное преимущество объекта PrintWriter перед обычными потоками символов заключается в том, что он ограждает вас от исключений, которые генерируются на уровне оригинальных потоков. В отличие от методов из других классов, методы объектов PrintWriter и PrintStream не генерируют исключения типа IOExceptions. Вместо этого они дают возможность проверять наличие ошибок. Это существенно упрощает такую распространенную операцию, как вывод текста. Вы можете выполнять проверку на ошибки при помощи метода checkError():

```
System.out.println( reallyLongString );
if ( System.out.checkError() ){ ... // ай ай ай
```

Конвейеры

Обычно приложения могут одновременно работать только с одним концом заданного потока. Тем не менее классы `PipedInputStream` и `PipedOutputStream` (или `PipedReader` и `PipedWriter`) позволяют создавать сразу оба конца, подключаясь к ним, как показано на рис. 12.4. С их помощью можно связывать, к примеру, разные потоки выполнения (не путать с потоками ввода/вывода), или они могут использоваться в качестве «заглушек» во время тестирования. Часто они играют роль переходника между двумя API-интерфейсами, один из которых поддерживает потоки, а другой нет.

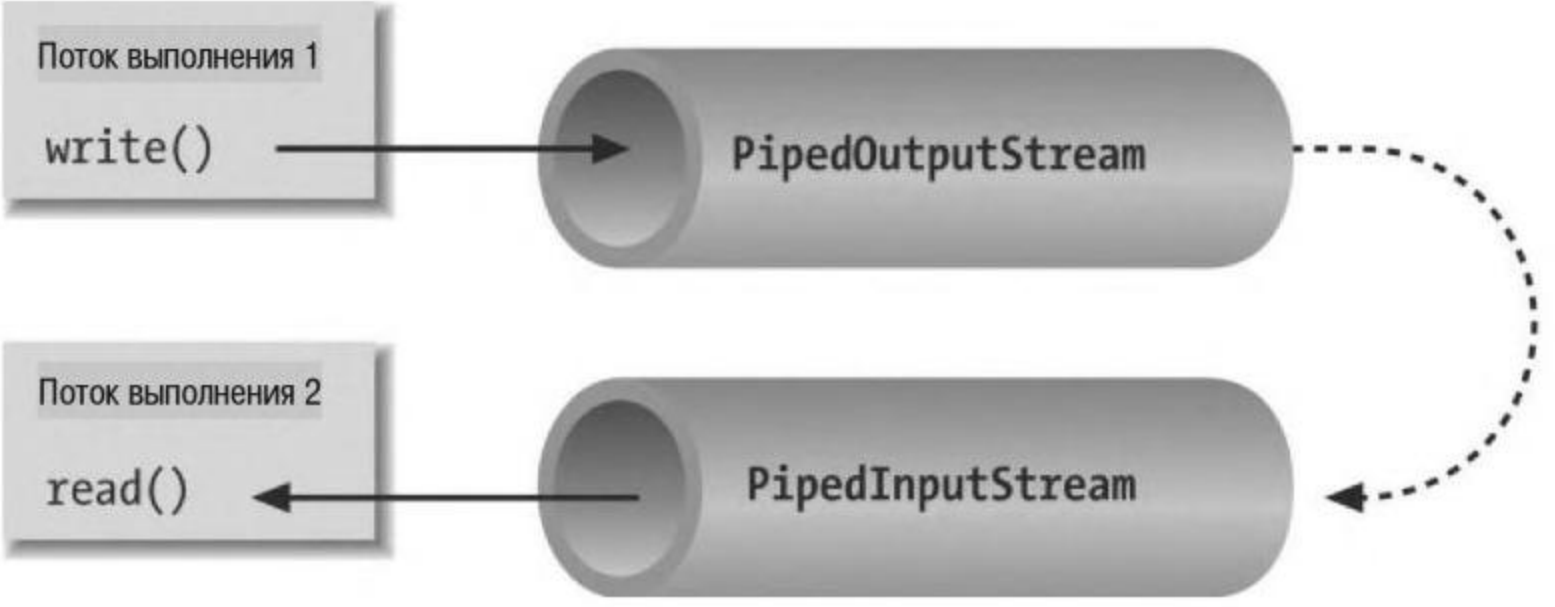

Рис. 12.4. Потоки на основе конвейеров

Чтобы создать конвейер для потока байтов, мы используем сразу два класса, `PipedInputStream` и `PipedOutputStream`. Сначала мы создаем первый конец, а потом используем его в качестве аргумента для создания второго:

```
PipedInputStream pin = new PipedInputStream();
PipedOutputStream pout = new PipedOutputStream( pin );
```

А можно и наоборот:

```
PipedOutputStream pout = new PipedOutputStream();
PipedInputStream pin = new PipedInputStream( pout );
```

В обоих примерах создаются связанные между собой входящий (`pin`) и исходящий (`pout`) потоки. Данные, записанные в `pout`, могут быть прочитаны из `pin`. Потоки `PipedInputStream` и `PipedOutputStream` можно создавать по отдельности, и затем соединять их с помощью метода `connect()`.

Мы можем сделать то же самое для символьных потоков, заменив классы `PipedInputStream` and `PipedOutputStream` на `PipedReader` и `PipedWriter` соответственно.

Соединив между собой оба конца конвейера, вы можете использовать их, как будто это два потока — входящий и исходящий. Вы можете считывать данные из объекта `PipedInputStream` (или `PipedReader`) с помощью метода `read()`, равно как записывать их в объект `PipedOutputStream` (или `PipedWriter`), используя метод `write()`. Когда буфер конвейера полностью заполняется, исходящий поток блокирует свою работу, ожидая появления свободного места. Входящий поток блокируется, когда конвейер становится пустым; он ждет, пока не станут доступны новые данные.

Одно из преимуществ от использования конвейеров заключается в том, что они предоставляют доступ к «поточным» функциям, не требуя от нас создания новых специализированных потоков. Чтобы показать, как это работает, мы создадим с помощью конвейеров простой компонент для ведения журнала внутри приложения (он же «консоль»). Используя обычный экземпляр `PrintWriter`, мы будем отправлять сообщения в журнал, где они при необходимости будут обрабатываться или буферизироваться, и затем доставляться к конечному адресату. Мы выбрали классы `PipedReader` и `PipedWriter`, так как здесь выполняется работа с текстовыми сообщениями. В следующем фрагменте кода показан каркас нашего компонента для ведения журнала:

```
class LoggerDaemon extends Thread
{
   PipedReader in = new PipedReader();
   LoggerDaemon() {
      start();
   }
   public void run() {
      BufferedReader bin = new BufferedReader( in );
      String s;
      try {
         while ( (s = bin.readLine()) != null ) {
            // обрабатываем строчку текста
         }
      } catch (IOException e ) { }
   }
   PrintWriter getWriter() throws IOException {
      return new PrintWriter( new PipedWriter( in ) );
```

```
    }
}
class myApplication {
    public static void main ( String [] args ) throws IOException {
        PrintWriter out = new LoggerDaemon().getWriter();
        out.println("Приложение запускается...");
        // ...
        out.println("Предупреждение: нельзя выполнить вычисление!");
        // ...
    }
}
```

Класс `LoggerDaemon` считывает данные из конца конвейера `PipedReader` под названием `in`. Он также предоставляет метод `getWriter()`, который возвращает экземпляр `PipedWriter`, подключенный к входящему потоку. Чтобы начать отправку сообщений, мы создаем новый объект `LoggerDaemon` и получаем исходящий поток. Чтобы читать строки с помощью метода `readLine()`, объект `LoggerDaemon` создает для конвейера `PipedReader` обертку в виде буферизированного потока `BufferedReader`. Для представления исходящего конвейера вместо обычного объекта `Writer` используется `PrintWriter` — это сделано в целях удобства.

Одно из преимуществ от использования конвейеров при реализации класса `LoggerDaemon` заключается в том, что записывать сообщения в журнал не сложнее, чем выводить текст в терминал или любой другой поток. Другими словами, мы можем использовать привычные нам инструменты и методики, включая метод `printf()`. Кроме того, для работы объекта `LoggerDaemon` выделяется отдельный поток выполнения; это означает, что пока он обрабатывает сообщения, мы можем заниматься своими делами.

Чтение и запись потоков строк

Еще одним полезным видом потоков является класс `StringReader`, который, в сущности, наделяет потоковыми возможностями объект типа `String`. Вот как его можно использовать:

```
String data = "В тридевятом царстве, в тридесятом государстве...";
StringReader sr = new StringReader( data );
char c1 = (char)sr.read(); // В
char c2 = (char)sr.read(); // пробел
char c3 = (char)sr.read(); // т
```

Стоит заметить, что здесь необходимо обрабатывать исключение IOExceptions, которое генерируется некоторыми методами класса StringReader.

Класс StringReader подходит для ситуаций, когда вам нужно читать данные из строки таким образом, как будто они поступают из потока (файла, конвейера или сокета). Допустим, у вас есть компонент для разбора данных, ориентированный на чтение из потока, но вы хотите добавить к нему еще один метод, который поддерживает большие строки. Это можно легко сделать с помощью класса StringReader.

Если подойти к проблеме с обратной стороны, класс StringWriter позволяет выполнять запись в буфер символов с помощью исходящего потока. При необходимости буфер расширяется, чтобы вместить данные. После завершения этого процесса мы можем извлечь содержимое буфера в виде объекта String. В следующем примере мы создаем экземпляр StringWriter и для удобства помещаем его внутрь объекта PrintWriter:

```
StringWriter buffer = new StringWriter();
PrintWriter out = new PrintWriter( buffer );
out.println("Однажды мою сестру укусил лось.");
out.println("Серьезно!");
String results = buffer.toString();
```

Сначала мы выводим несколько строчек в исходящий поток, просто чтобы наполнить его какими-то данными, и затем извлекаем результат в виде строки при помощи метода toString(). Как вариант, мы могли бы воспользоваться методом getBuffer(), чтобы вместо строки получить объект типа StringBuffer.

Класс StringWriter может быть полезен в ситуациях, когда вы хотите получить вывод из компонента, который обычно отправляет данные в поток (файл или консоль). Объект PrintWriter, внутрь которого заключен экземпляр StringWriter, можно использовать для последовательного создания больших строк вместо буфера StringBuffer.

По тому же принципу работают классы ByteArrayInputStream и ByteArrayOutputStream, только вместо символов они ориентированы на байты. Вы можете записывать байты в поток ByteArrayInputStream, и затем извлекать их с помощью метода toByteArray(). Можно сделать наоборот: получить поток ByteArrayInputStream из массива байтов, как мы это делали с объектом StringReader и строками. К примеру, чтобы посмотреть, что именно записывает объект DataOutputStream во время

кодирования конкретного значения, мы можем воспользоваться исходящим потоком массива байтов:

```
ByteArrayOutputStream bao = new ByteArrayOutputStream();
DataOutputStream dao = new DataOutputStream( bao );
dao.writeInt( 16777216 );
dao.flush();
byte [] bytes = bao.toByteArray();
for( byte b : bytes )
   System.out.println( b ); // 1, 0, 0, 0
```

Реализация фильтрующего потока

Прежде чем переходить к следующей теме, давайте создадим свой собственный поток. Ранее мы уже упоминали, что в основе узкоспециализированных потоков-оберток лежат классы `FilterInputStream` и `FilterOutputStream`. Не так уж сложно создать производную от `FilterInputStream`, которая могла бы служить оберткой для других потоков, добавляя в них новые функции.

Следующий пример, `rot13InputStream`, считывает байты и выполняет для них операцию *rot13* (сдвиг на 13 букв). Это тривиальный алгоритм, который сдвигает буквы таким образом, чтобы человек не смог прочитать полученный результат (неалфавитные символы просто игнорируются). Несмотря на свою простоту, метод `rot13` обладает определенным изяществом: он симметричен, поэтому для того, чтобы вернуть тексту его первоначальный вид, достаточно пропустить его через этот метод во второй раз. Вот код класса `rot13InputStream`:

```
public class rot13InputStream extends FilterInputStream
{
   public rot13InputStream ( InputStream i ) {
      super( i );
   }
   public int read() throws IOException {
      return rot13( in.read() );
   }
   // нужно переопределить остальные версии метода read()
   private int rot13 ( int c ) {
      if ( (c >= 'A') && (c <= 'Z') )
         c=(((c-'A')+13)%26)+'A';
```

```
        if ( (c >= 'a') && (c <= 'z') )
            c=(((c-'a')+13)%26)+'a';
        return c;
    }
}
```

Объект `FilterInputStream` нужно инициализировать с помощью аргумента типа `InputStream` (это поток, который будет фильтроваться). Внутри конструктора, который мы предоставляем для класса `rot13InputStream`, необходимо вызвать метод `super()`, который выполнит родительский конструктор. Класс `FilterInputStream` содержит защищенное поле `in`, которое хранит ссылку на заданный экземпляр `InputStream`, делая его доступным для остальной части нашего класса.

Главная особенность класса `FilterInputStream` заключается в том, что он делегирует свои входящие задачи оригинальному потоку `InputStream`. К примеру, вызов метода `read()` из объекта `FilterInputStream` приводит к вызову того же метода из объекта `InputStream`, благодаря чему байты извлекаются из исходного потока. Фильтрация происходит на этапе, когда мы выполняем обработку данных, проходящих через поток. В нашем примере метод `read()` извлекает байты из переменной `in` (типа `InputStream`), и, прежде чем их возвращать, применяет к ним алгоритм rot13. Метод `rot13()` сдвигает буквы, перебирая все значения подряд, включая идентификатор окончания потока (–1). Таким образом, наш дочерний класс является фильтром на основе алгоритма rot13.

`read()` — это единственный метод класса `InputStream`, который мы переопределяем внутри `FilterInputStream`. Остальные возможности потока `InputStream` остаются неизменными, поэтому на любые другие вызовы, такие как `skip()` или `available()`, он будет отвечать самостоятельно.

Строго говоря, класс `rot13InputStream` работает только с теми байтами, которые закодированы в формате ASCII, так как алгоритм сдвига основан на латинском алфавите. Чтобы поддерживать более широкий набор символов, нам бы пришлось работать с 16-битной кодировкой Unicode; для этого в качестве базового класса для нашего алгоритма нужно было бы выбрать `FilterReader` (хотите попробовать rot32768?). Стоит также отметить, что мы не до конца реализовали наш фильтр: нам следовало бы переопределить версию метода `read()`, которая принимает массив байтов и набор модификаторов; возможно, пришлось бы делегировать ее вызовы нашей собственной реализации `read`. Но так как мы этого не сделали, данный метод просто возвращает исходный поток.

Файловый ввод/вывод

Эта глава посвящена API-интерфейсу для файлового ввода/вывода. Если точнее, мы обсудим два таких интерфейса. Один из них входит в состав платформы Java с момента ее появления и считается основным средством ввода/вывода в пакете `java.io`. Существует также «новый» интерфейс; он был представлен вместе с выходом платформы Java 7 и находится в пакете `java.nio.file`. В целом, пакеты в составе технологии NIO (позже мы займемся ее подробным рассмотрением, затрагивая не только файлы, но также сеть и каналы) создавались для того, чтобы сделать платформу Java более масштабируемой и производительной. Однако функциональность, связанная с файловым вводом/выводом, во многом перекликается со старым API-интерфейсом. Говоря языком киноиндустрии, мы имеем дело с «классикой» и «ремейком». Новый API-интерфейс по своим функциям полностью дублирует старый, но так как инструменты, входящие в состав пакета `java.io`, являются фундаментальными (а в некоторых случаях и более простыми), многие разработчики предпочитают и дальше использовать именно их. Мы начнем с классического API-интерфейса, основанного на классе `java.io.File`, после чего перейдем к его новому аналогу, `java.nio.Path`.

Работать с файлами на платформе Java довольно легко, однако здесь вы можете столкнуться с некоторыми проблемами концептуального характера. Реальные файловые системы могут сильно отличаться друг от друга в плане архитектуры и реализации: достаточно вспомнить о разнице в подходах к именованию файлов в системах Mac, Windows и Unix. Платформа Java пытается скрыть некоторые различия, предоставляя информацию, которая помогает приложениям приспосабливаться к локальной среде. Тем не менее многие подробности, касающиеся доступа к файлам, отдаются на откуп конкретной реализации. По мере чтения этой главы вы познакомитесь с некоторыми методиками для решения вышеописанных проблем.

В конце этого раздела мы покажем вам некоторые инструменты для работы с особым видом «ресурсных» файлов, которые упаковываются вместе с приложением и загружаются по тому же пути, что и все классы в Java.

Класс `java.io.File`

Класс `java.io.File` инкапсулирует доступ к информации о файле или директории. С его помощью можно получить данные об атрибутах файла, список записей в директории, а также выполнять некоторые про-

стейшие операции с файловой системой, такие как удаление файла или создание директории. Объект `File` отвечает за операции общего назначения, не предоставляя API-интерфейса для чтения или записи содержимого файлов; для этих целей существуют файловые потоки.

Конструкторы класса `File`

Вы можете создать экземпляр класса `File`, указав путь к файлу или директории:

```
File fooFile = new File( "/tmp/foo.txt" );
File barDir = new File( "/tmp/bar" );
```

Файл можно создать и с помощью относительного пути:

```
File f = new File( "foo" );
```

В этом случае путь будет сформирован относительно «текущей рабочей директории» Java-машины. Эту директорию можно определить, прочитав запись `user.dir` в списке системных свойств:

```
System.getProperty("user.dir"); // например,"/Users/pat"
```

Перегруженная версия конструктора `File` позволяет указывать путь к директории и имя файла в виде двух отдельных строк:

```
File fooFile = new File( "/tmp", "foo.txt" );
```

У этого конструктора есть еще одна разновидность, которая принимает директорию в виде объекта `File`, а имя файла — как строку:

```
File tmpDir = new File( "/tmp" ); // объект File для директории /tmp
File fooFile = new File ( tmpDir, "foo.txt" );
```

Ни один из этих конструкторов физически ничего не создает — он может указывать на несуществующий файл, и это не будет ошибкой. Объект `File` — это всего лишь идентификатор файла или директории, чьи свойства вам бы хотелось прочитать, записать или проверить. К примеру, с помощью метода `exists()` вы можете узнать, существует ли файл или директория.

Адаптация путей в разных системах

При работе с файлами на платформе Java неизменно возникает одна проблема: в именах путей должны соблюдаться условности локальных файловых систем. Операционная система Windows использует концепцию дисков, которые представляют «корневые точки» (например, *C:*);

в качестве разделителей в ней применяются обратные слеши (\) вместо прямых (/). Этим она отличается от другим систем.

Платформа Java пытается нивелировать эти различия. Например, в ОС семейства Windows в путях допускаются как прямые, так и обратные слеши (хотя в остальных системах можно использовать только прямые).

Самый лучший вариант — соблюдать условности, принятые в файловой системе, с которой вы работаете. Если у вашего приложения есть графический пользовательский интерфейс, который позволяет открывать и сохранять файлы, то все эти функции можно переложить на класс `JFileChooser` из библиотеки Swing. Этот класс инкапсулирует графическое диалоговое окно выбора файлов. Его методы сами позаботятся об особенностях именования файлов, зависящих от конкретной системы.

Но все становится немного сложнее, если вашему приложению нужно самостоятельно работать с файлами. Для выполнения задач подобного рода в классе `File` предусмотрено несколько статических переменных. В поле `File.separator` хранится строка, в которой указан разделитель, актуальный для текущей системы (например, в Unix и Mac это будет «/», а в системах семейства Windows — «\»); поле `File.separatorChar` предоставляет ту же информацию, только в виде символа (`char`).

Эти переменные можно использовать по-разному. Чтобы адаптировать локальные пути, проще всего, наверное, будет установить для себя следующее правило: внутри программы используются прямые слеши (/), а при работе с файловой системой они заменяются локальными разделителями:

```
// в качестве стандарта мы будем использовать прямой слеш
String path = "mail/2004/june/merle";
path = path.replace('/', File.separatorChar);
File mailbox = new File( path );
```

Вы также можете разбивать путь на отдельные составляющие, при необходимости собирая его локальную версию:

```
String [] path = { "mail", "2004", "june", "merle" };
StringBuffer sb = new StringBuffer(path[0]);
for (int i=1; i< path.length; i++)
   sb.append( File.separator + path[i] );
File mailbox = new File( sb.toString() );
```

Здесь нужно кое-что запомнить: если обратный слеш (\) указан в исходном коде внутри строкового значения, он интерпретируется платформой Java как управляющий символ. Чтобы получить обратный слеш внутри строки, используйте запись «\\».

Чтобы решить проблему нескольких «корневых точек» (например, *C:* в Windows), класс `File` предоставляет статический метод `listRoots()`, который возвращает массив объектов `File`, представляющих корневые директории файловой системы. Опять же, в приложениях с графическим интерфейсом эта проблема полностью решается за счет диалогового окна выбора файлов.

Операции с файлами

Создав объект `File`, мы можем использовать его для получения информации о файлах и директориях, а также для выполнения стандартных операций с файловой системой. Есть методы, которые позволяют задавать вопросы. Например, методы `isFile()` и `isDirectory()` возвращают `true`, если объект `File` представляет собой, соответственно, обычный файл или директорию. Метод `isAbsolute()` позволяет узнать, какой путь используется в объекте `File` — *полный* или *относительный*. В разных системах полный путь записывается по-разному, однако он никак не зависит от текущей рабочей директории приложения или концепции «корневых точек» (дисков). Например, в операционной системе Windows полный путь содержит букву диска: *c:\\Users\pat\foo.txt*.

Составные части пути в объекте `File` доступны посредством следующих методов: `getName()`, `getPath()`, `getAbsolutePath()` и `getParent()`. Метод `getName()` возвращает строку с именем файла, без какой-либо информации о директории. Если в объекте `File` указан полный путь, его можно получить с помощью метода `getAbsolutePath()`; в противном случае будет возвращен относительный путь, добавленный к текущей рабочей директории (в попытке сформировать полный путь). Метод `getParent()` возвращает родительскую директорию файла или каталога.

Строка, возвращаемая методами `getPath()` и `getAbsolutePath()`, может не соответствовать условностям, которые приняты в исходной файловой системе и касаются регистра в именах файлов и директорий. Чтобы получить «каноническую» версию пути, предоставляемую самой файловой системой, используйте метод `getCanonicalPath()`. К примеру, в Windows вы можете создать объект `File`, метод `getAbsolutePath()` которого возвращает *C:\Autoexec.bat*, в то время как результатом выпол-

нения метода `getCanonicalPath()` будет *C:\AUTOEXEC.BAT* (хотя оба пути указывают на один и тот же файл). Это может пригодиться при сравнении имен файлов, которые были сформированы с учетом разных соглашений об именовании; вы также можете использовать эти методы для вывода путей на экран.

С помощью методов `lastModified()` и `setLastModified()`можно считывать и записывать дату изменения файла или директории. Значение даты имеет тип `long` и хранит количество миллисекунд, прошедших с начала так называемой *эры Unix* (1 января 1970 года, 00:00:00 GMT). Мы также можем получить размер файла в байтах, вызвав метод `length()`.

Ниже представлен фрагмент кода, который выводит определенную информацию о файле:

```
File fooFile = new File( "/tmp/boofa" );
String type = fooFile.isFile() ? "Файл " : "Директория ";
String name = fooFile.getName();
long len = fooFile.length();
System.out.println( type + name + ", " + len + " байтов " );
```

С помощью методов `list()` и `listFiles()` можно получить список файлов в директории, на которую указывает объект `File`:

```
File tmpDir = new File("/tmp" );
String [] fileNames = tmpDir.list();
File [] files = tmpDir.listFiles();
```

Метод `list()` возвращает массив объектов `String` с именами файлов. Метод `listFiles()` возвращает массив объектов `File`. Следует отметить, что ни один из этих вариантов не гарантирует того, что файлы будут размещены в каком-либо порядке (например, алфавитном). Чтобы отсортировать файлы по их именам, можно воспользоваться API-интерфейсом для работы с коллекциями:

```
List list = Arrays.asList( sa );
Collections.sort(list);
```

Если директория, на которую указывает объект `File`, не существует, мы можем ее создать, используя метод `mkdir()` или `mkdirs()`. Метод `mkdir()` создает только одну директорию, без вложенных уровней, поэтому все промежуточные директории, входящие в состав пути, долж-

ны существовать изначально. Метод `mkdirs()` позволяет при необходимости создать дерево директорий, которое соответствует полному пути, указанному в объекте `File`. Если директорию создать не удается, в обоих случаях возвращается `false`. Чтобы переименовать или удалить файл или директорию, используйте методы `renameTo()` и `delete()` соответственно.

С помощью объекта `File` можно создать директорию, но для создания файлов этот способ используется не так часто; обычно мы создаем файлы в том момент, когда собираемся записывать данные при помощи объектов `FileOutputStream` или `FileWriter` (об этом мы скоро поговорим). Исключение составляет метод `createNewFile()`, который используется для создания нового файла нулевой длины (путь к файлу указывается посредством объекта `File`). У этого метода есть одна полезная особенность: он гарантирует «атомарное» выполнение операций, учитывая любые другие процедуры создания файлов, которые происходят в рамках файловой системы. Метод `createNewFile()` возвращает булево значение, по которому можно понять, удалось ли нам создать файл. Иногда это используется в качестве примитивного механизма блокирования — кто начал создание первым, тот и «выигрывает». (Позже вы увидите, что пакет NIO поддерживает настоящее блокирование фалов.) Вы можете использовать этот подход в сочетании с методом `deleteOnExit()`, который маркирует файл для автоматического удаления после завершения работы Java-машины; это поможет вам беречь ресурсы и создавать приложения, которые можно запускать исключительно в единственном экземпляре. Класс `File` предоставляет еще один статический метод `createTempFile()`, который создает файлы в специальной директории, используя уникальные имена, сгенерированные автоматически. Он тоже может быть полезен в связке с методом `deleteOnExit()`.

Метод `toURL()` преобразует путь к файлу в объект `URL`. URL — это абстракция, которая позволяет ссылаться на файл, находящийся в любой точке сети. Такое преобразование позволяет добиться совместимости с более универсальными инструментами, которые оперируют адресами в формате URL (подробности в главе 14). Этот формат широко используется в API-интерфейсе для работы с файлами из пакета NIO, где с его помощью обеспечивается адресация в новых файловых системах, реализованных непосредственно на языке Java.

В табл. 12.1 подытожен список методов, которые предоставляет класс `File`.

Табл. 12.1. Методы класса `File`

Метод	Возвращаемый тип	Описание
`canExecute()`	`boolean`	Является ли файл исполняемым?
`canRead()`	`boolean`	Доступен ли файл (или директория) для чтения?
`canWrite()`	`boolean`	Доступен ли файл (или директория) для записи?
`createNewFile()`	`boolean`	Создает новый файл.
`createTempFile (String pfx, String sfx)`	`File`	Статический метод для создания новых файлов во временной директории с возможностью задания префикса и суффикса.
`delete()`	`boolean`	Удаляет файл (или директорию)
`deleteOnExit()`	`Void`	Java-машина удаляет файл по завершению работы.
`exists()`	`boolean`	Существует ли файл (или директория)?
`getAbsolutePath()`	`String`	Возвращает полный путь к файлу (или директории).
`getCanonicalPath()`	`String`	Возвращает полный, корректный с точки зрения регистра путь к файлу (или директории).
`getFreeSpace()`	`long`	Возвращает количество байтов неиспользованного пространства на разделе, куда указывает путь (или 0, если путь недействителен).
`getName()`	`String`	Возвращает имя файла (или директории).
`getParent()`	`String`	Возвращает имя родительской директории файла (или директории).
`getPath()`	`String`	Возвращает путь к файлу (или директории). Не путать с `toPath()`.
`getTotalSpace()`	`long`	Возвращает размер раздела, на который указывает путь (в байтах) или 0, если путь недействителен.
`getUseableSpace()`	`long`	Возвращает количество байтов неиспользованного, но доступного пользователю пространства на разделе, куда указывает путь (или 0, если путь недействителен). Этот метод пытается учитывать права на запись, которые имеет пользователь.
`isAbsolute()`	`boolean`	Является ли имя файла (или директории) абсолютным?
`isDirectory()`	`boolean`	Является ли этот элемент директорией?
`isFile()`	`boolean`	Является ли этот элемент файлом?

Метод	Возвращаемый тип	Описание
`isHidden()`	`boolean`	Является ли элемент скрытым (зависит от системы).
`lastModified()`	`long`	Возвращает время последнего изменения файла (или директории).
`length()`	`long`	Возвращает длину файла.
`list()`	`String[]`	Возвращает список файлов в директории.
`listFiles()`	`File[]`	Возвращает содержимое директории в виде массива объектов `File`.
`listRoots()`	`File[]`	Возвращает корневые точки системы, если таковые имеются (например, *C:/*, *D:/*).
`mkdir()`	`boolean`	Создает директорию.
`mkdirs()`	`boolean`	Создает все директории, из которых состоит путь.
`renameTo(File` *dest* `)`	`boolean`	Переименовывает файл (или директорию).
`setExecutable()`	`boolean`	Устанавливает права на выполнение файла.
`setLastModified()`	`boolean`	Устанавливает время последнего изменения файла (или директории).
`setReadable()`	`boolean`	Устанавливает права на чтение файла.
`setReadOnly()`	`boolean`	Делает файл доступным только для чтения.
`setWriteable()`	`boolean`	Устанавливает права на запись файла.
`toPath()`	`java.nio.file.Path`	Конвертирует экземпляр `File` в объект `Path` из пакета NIO (см. раздел «API-интерфейс пакета NIO для работы с файлами»). Не путать с `getPath()`.
`toURL()`	`java.net.URL`	Генерирует объект `URL` для файла (или директории).

Файловые потоки

Вам уже, наверное, надоело читать обо всех этих файлах, а ведь мы еще не записали ни единого байта! Ну что ж, пришло время для веселья. Платформа Java предоставляет два фундаментальных потока для чтения и записи файлов: `FileInputStream` и `FileOutputStream`. Являясь наследниками классов `InputStream` и `OutputStream`, эти потоки обладают базовыми функциями для работы с байтами, которые относятся к чтению и записи данных на диск. Их можно использовать в сочетании

с фильтрующими потоками, описанными ранее, чтобы работа с файлами не отличалась от любого другого потокового взаимодействия.

Поток `FileInputStream` можно создать из строки, хранящей путь, или из объекта `File`:

```
FileInputStream in = new FileInputStream( "/etc/passwd" );
```

Когда вы создаете объект `FileInputStream`, Java-машина автоматически пытается открыть указанный файл. Из-за этого конструкторы `FileInputStream` могут генерировать исключение `FileNotFoundException`, если такого файла не существует, или `IOException`, если произошла другая ошибка ввода/вывода.

Вы должны отлавливать эти исключения в своем коде. Мы рекомендуем использовать новую конструкцию try-with-resources из состава Java 7 везде, где только можно; она автоматически закрывает файлы, когда вы заканчиваете с ними работать:

```
try ( FileInputStream fin = new FileInputStream( "/etc/passwd" ) ) {
   ....
   // При выходе из блока try файл fin автоматически
   // закроется, если в этом будет необходимость.
}
```

Когда поток создается впервые, его метод `available()` должен возвращать то же значение, что и метод `length()` из объекта `File`.

Чтобы читать символы из файла, как это делает объект `Reader`, вы можете поместить поток `FileInputStream` внутрь экземпляра `InputStreamReader`. В сущности, такую же комбинацию вы могли бы получить при помощи класса `FileReader`, однако у этого решения есть один существенный недостаток: по какой-то необъяснимой причине данный класс использует исключительно стандартную системную кодировку, которую нельзя поменять, поэтому вам лучше остановиться на варианте с объектами `FileInputStream` и `InputStreamReader`.

Ниже представлен класс `ListIt`; это небольшая утилита, которая отправляет содержимое файла или директории в стандартный вывод:

```
//файл: ListIt.java
import java.io.*;
class ListIt {
public static void main ( String args[] ) throws Exception {
```

```
File file = new File( args[0] );
if ( !file.exists() || !file.canRead() ) {
System.out.println( "Can't read " + file );
return;
}
if ( file.isDirectory() ) {
String [] files = file.list();
for ( String file : files )
System.out.println( file );
} else
try {
Reader ir = new InputStreamReader(
new FileInputStream( file ) );
BufferedReader in = new BufferedReader( ir );
String line;
while ((line = in.readLine()) != null)
File I/O | 431
System.out.println(line);
}
catch ( FileNotFoundException e ) {
System.out.println( "Файл исчез" );
}
}
}
```

Класс `ListIt` использует первый аргумент командной строки, чтобы создать объект `File`, после чего проверяет, существует ли такой файл, и доступен ли он для чтения. Если объект `File` представляет директорию, на экран выводится список файлов, которые в ней хранятся. В противном случае класс `ListIt` считывает файл и выводит его содержимое, строчка за строчкой.

Вы можете создать поток `FileOutputStream` для записи файлов, используя объект `File` или строку, хранящую путь. Однако в отличие от `FileInputStream`, конструкторы класса `FileOutputStream` не генерируют исключение `FileNotFoundException`.

Если указанного файла не существует, объект `FileOutputStream` создаст его сам. Тем не менее, если случится какая-то другая ошибка, связанная с вводом/выводом, конструкторы этого класса могут сгенерировать исключение `IOException`, поэтому вам нужно предусмотреть обработку таких ситуаций.

Если указанный файл существует, поток `FileOutputStream` открывает его для записи. Затем, когда вы вызовите метод `write()`, текущее содержимое файла будет перезаписано новыми данными.

Если вы хотите добавлять данные в конец файла, не затрагивая ранее записанную информацию, вы можете использовать версию конструктора, которая принимает аргумент типа `Boolean`:

```
// перезаписываем файл fooFile
FileInputStream fooOut =
   new FileOutputStream( fooFile );
// добавляем данные в конец
FileInputStream pwdOut =
   new FileOutputStream( "/etc/passwd", true );
```

Добавлять данные в файл можно также при помощи класса `RandomAccessFile`, который мы скоро обсудим.

Как и в случае с чтением, чтобы записывать в файл символы (а не байты), вам нужно заключить поток `FileOutputStream` внутрь объекта `OutputStreamWriter`. Вместо этого вы можете воспользоваться удобным классом `FileWriter`, если вам подходит стандартная системная кодировка.

В следующем примере данные построчно считываются из стандартного ввода и записываются в файл */tmp/foo.txt*:

```
String s = new BufferedReader(
   new InputStreamReader( System.in ) ).readLine();
File out = new File( "/tmp/foo.txt" );
FileWriter fw = new FileWriter ( out );
PrintWriter pw = new PrintWriter( fw )
pw.println( s );
pw.close();
```

Обратите внимание, что мы поместили поток `FileWriter` внутрь объекта `PrintWriter`, сделав тем самым процесс записи более удобным. Чтобы соблюсти «правила приличия», принятые в файловой системе, по завершении работы с потоком `FileWriter` мы вызвали метод `close()`.

Вместе с объектом `PrintWriter` закрывается и его исходный поток. Чтобы не закрывать файл вручную, мы могли бы использовать конструкцию try-with-resources.

Класс `RandomAccessFile`

Класс `java.io.RandomAccessFile` позволяет считывать и записывать данные на произвольном участке файла. Он реализует сразу два интерфейса, `DataInput` и `DataOutput`, поэтому вы можете использовать его для чтения и записи строк и простых типов данных, относясь к файлам, как к обычным потокам типа `DataInputStream` или `DataOutputStream`. Но ввиду того, что этот класс поддерживает произвольный, а не последовательный доступ к содержимому файлов, он не является наследником классов `InputStream` и `OutputStream`.

Вы можете создать экземпляр `RandomAccessFile` на основе объекта `File` или строки, содержащей путь. У конструктора также есть второй аргумент типа `String`, который определяет режим доступа к файлу. Если вам нужно только чтение, используйте модификатор «`r`»; если вы хотите и читать, и записывать, укажите значение «`rw`».

```
try {
   RandomAccessFile users = new RandomAccessFile( "Users", "rw" )
} catch (IOException e) { ... }
```

Если вы создаете объект `RandomAccessFile` в режиме только для чтения и указанный вами файл не существует, генерируется исключение `IOException`. Но если использовать режим чтения *и* записи, файл будет создан автоматически. В любом случае вам нужно обрабатывать исключение `IOException`, так как оно может быть сгенерировано в результате любой другой ошибки ввода/вывода.

Создав объект `RandomAccessFile`, вы сможете вызывать из него любые методы чтения и записи, как вы это делали с потоками `DataInputStream` и `DataOutputStream`. Если вы попытаетесь выполнить запись в файл, предназначенный только для чтения, будет сгенерировано исключение `IOException`.

Но главной особенностью класса `RandomAccessFile` является метод `seek()`. Значение типа `long`, которое он принимает, служит смещением (в байтах) для выбора конкретного места в файле, где нужно выполнить запись или чтение. Чтобы получить текущую позицию, используйте метод `getFilePointer()`. Если вам нужно записать данные в конец файлы, вызовите сначала метод `length()`, чтобы определить местоположение последнего байта, и затем перейдите туда с помощью метода `seek()`. Вы можете пересечь конец файла и записать данные в позиции, которая на-

ходится за его пределами, но если при этом вы попытаетесь что-то прочитать, метод `read()` сгенерирует исключение `EOFException`.

Ниже показан пример записи информации в простейшую базу данных:

```
users.seek( userNum * RECORDSIZE );
users.writeUTF( userName );
users.writeInt( userID );
...
```

В этом довольно нехитром примере мы исходим из того, что длина строковой переменной `userName`, равно как и любого другого фрагмента данных, который будет записан после нее, не превышает заданный размер (`RECORDSIZE`).

Пути к ресурсам

Не последнее место в процессе упаковки и установки приложения занимает работа с ресурсами, такими как конфигурационные и графические файлы, а также программные данные. Платформа Java предлагает несколько способов для доступа к этим ресурсам. Например, можно просто открывать файлы и считывать из них байты. Еще один вариант: формировать URL-адреса, которые указывают на хорошо известные директории в файловой системе или сети (подробности работы с URL-адресами мы обсудим в главе 14). Проблема в том, что оба эти подхода предполагают использование сведений о местоположении программы и ее пакетов; эта информация может быть изменена или даже утеряна, если файлы программы будут перемещены. Нам нужно по-настоящему универсальное средство доступа к ресурсам приложения, независимо от того, как они были установлены. В качестве такого средства выступает метод `getResource()` из класса `Class`, который используется в сочетании с параметром (или переменной среды) classpath. Например:

```
URL resource = MyApplication.class.getResource("/config/config.xml");
```

Вместо того чтобы создавать объект `File`, ссылающийся на полный путь, или полагаться на сведения об установочной директории, вы можете воспользоваться методом `getResource()`, который предоставляет стандартный способ получения ресурсов. Местонахождение ресурсов может определяться либо относительно заданного class-файла, либо с помощью общей системной директории classpath. Для получения данных метод `getResource()` использует загрузчик классов, с помощью ко-

торого загружаются class-файлы приложения. Это означает, что где бы ни находились ваши классы — на веб-сервере, в файловой системе или даже внутри JAR-файла или другого архива — вы всегда можете загрузить ресурсы, которые упакованы вместе с ними.

И хотя мы пока не дошли до обсуждения URL-адресов, уже сейчас можно сказать, что многие API-интерфейсы для загрузки данных (например, изображений) напрямую принимают пути именно в этом формате. Считывая данные самостоятельно, вы можете вызвать метод `openStream()`, чтобы получить из URL-адреса объект `InputStream`, с которым можно работать как с любым другим потоком. Чтобы пропустить этот этап и получить поток `InputStream` напрямую, можно воспользоваться удобным методом `getResourceAsStream()`.

Метод `getResource()` принимает в качестве аргумента *путь к ресурсу* (разделенный слешами), возвращая в ответ его URL-адрес. У ресурсов бывает два вида путей — полные и относительные. Полный путь начинается со слеша (например, */config/config.xml*). В этом случае поиск заданного файла стартует в самом «верху» иерархии classpath (директории или JAR-архива). Если указать путь */config/config.xml*, то в процессе поиска будет проверена каждая директория и JAR-архив, пока не будет найден файл *config/config.xml*. При этом не важно, из какого именно класса был вызван метод `getResource()`; главное, чтобы файл ресурса содержался в директории classpath, принадлежащей загрузчику этого класса. Например:

```
URL data = AnyClass.getResource("/config/config.xml");
```

Относительные URL-адреса не начинаются со слеша (например, *mydata.txt*). В этом случае начальной точкой поиска является местонахождение class-файла, в котором был вызван метод `getResource()`. Другими словами, путь берется относительно пакета, в котором находится наш файл с классом. Например, если класс `foo.bar.MyClass` имеет путь *foo/bar/MyClass.class*, то для загрузки файла *mydata.txt*, который находится в той же директории или JAR-архиве в рамках classpath (*foo/bar/mydata.txt*), достаточно выполнить следующий код:

```
URL data = MyClass.getResource("mydata.txt");
```

Здесь класс и файл загружаются из одной логической директории. Мы говорим «логической», потому что поиск не ограничивается тем участком classpath, из которого был загружен класс. Ресурс ищется во всех местах classpath (как и в случае с полным путем), пока не будет найден. И хотя можно было бы предположить, что файл *mydata.txt* упа-

кован вместе с классом *MyClass.class*, он может быть найден и в другом JAR-архиве или директории — главное, чтобы он был в пределах того же относительного пути.

Ниже представлен пример приложения, которое подключает некоторые ресурсы:

```
package mypackage;
import java.net.URL;
import java.io.IOException;
public class FindResources {
   public static void main( String [] args ) throws IOException {
      // полный путь, полученный с помощью classpath
      URL url = FindResources.class.getResource("/mypackage/foo.txt");
      // получение ресурса относительно местонахождения класса
      url = FindResources.class.getResource("foo.txt");
      // еще один документ с относительным путем
      url = FindResources.class.getResource("docs/bar.txt");
   }
}
```

Класс `FindResources` принадлежит пакету `mypackage`, поэтому его class-файл находится в директории *mypackage*, где-то внутри classpath. Файл *foo.txt* ищется сначала с помощью полного, а потом относительного URL-адреса. В самом конце относительный путь используется для получения документа в директории *mypackage/docs*. В каждом из этих случаев мы используем статическую переменную `class`, чтобы сослаться на объект `Class` нашего класса `FindResources`. Если бы у нас был экземпляр `FindResources`, мы могли бы вызвать из него метод `getClass()`, чтобы получить объект `Class`.

Опять же, метод `getResource()` возвращает URL-адрес для объекта того типа, который вы указали. Это может быть текстовый файл или файл свойств, который вы хотите прочитать в виде потока; это также может быть звуковой или графический файл или какой-то другой объект. Вы можете получить поток из URL-адреса, чтобы самостоятельно обработать данные, или передать объект `URL` в API-интерфейс, который сделает все за вас. Мы подробно обсудим URL-адреса в главе 14. Стоит также отметить, что, загружая ресурсы подобным образом, вы не обязаны знать о том, как именно они были упакованы или установлены. Даже если файлы приложения упаковать в JAR-архив, его ресурсы все равно будут загружаться. Java-апплеты (которые будут рассмотрены в гла-

ве 23) могут использовать этот подход для загрузки файлов по сети, так как загрузчик их классов воспринимает сервер как часть своего пути classpath.

API-интерфейс пакета NIO для работы с файлами

Мы закончили с «классическим» API-интерфейсом для работы с файлами и теперь можем сосредоточиться на альтернативных компонентах, которые предлагает новый пакет NIO, появившийся в Java 7. Как мы уже упоминали ранее, инструменты для работы с файлами из состава пакета NIO (`java.nio.file`) можно рассматривать как замену или дополнение классического API-интерфейса. Они являются частью инициативы, которая была призвана улучшить производительность платформы Java, сделать ее средства ввода/вывода более гибкими (что подразумевает поддержку *выборочности* и асинхронного прерывания *каналов*). Но в контексте работы с файлами главным преимуществом нового API-интерфейса является более углубленное абстрагирование от реальных *файловых систем*.

Помимо улучшенной поддержки имеющихся файловых систем (например, впервые появилась возможность копировать и перемещать файлы, управлять ссылками, получать подробные атрибуты файлов, связанных с их владельцами и правами доступа) технология NIO позволяет создавать на языке Java совершенно новые ФС. Ярчайшим примером этому является возможность «подключать» файлы ZIP-архивов и работать с их содержимым, как с обычной файловой системой, используя стандартные API-интерфейсы. К тому же пакет `java.nio.file` предоставляет некоторые инструменты, способные сохранить разработчикам множество времени на написание постоянно повторяющихся операций, включая слежение за изменениями в дереве директорий, обход файловой системы (шаблон проектирования «посетитель»), «универсализацию» файловых имен, а также методы для считывания всего содержимого файлов непосредственно в память.

В этом разделе мы сосредоточимся на основном API-интерфейсе для работы с файлами, а в конце главы опять вернемся к пакету NIO, чтобы подробно обсудить буферы и каналы, которые предоставляет эта технология. В частности, мы поговорим об интерфейсах `ByteChannels` и `FileChannel`, которые можно считать альтернативой буферизированным потокам для чтения/записи файлов и других типов данных.

Классы `FileSystem` и `Path`

Пакет `java.nio.file` состоит из трех главных компонентов. Класс `FileSystem` —базовый механизм хранения данных, который играет роль фабрики для объектов `Path`. Класс `Path` представляет файлы и директории в рамках файловой системы. Третий компонент, класс `Files`, является аналогом классического API-интерфейса; он предлагает богатый выбор статических методов для управления объектами `Path`, позволяя выполнять все основные операции с файлами.

Нашей отправной точкой станет класс `FileSystems` (с «s» в конце), который является фабрикой для объектов `FileSystem`:

```
// Стандартная файловая система компьютера
FileSystem fs = FileSystems.getDefault();
// Файловая система, выбранная вручную
URI zipURI = URI.create("jar:file:/Users/pat/tmp/MyArchive.zip");
FileSystem zipfs = FileSystems.newFileSystem( zipURI, env ) );
```

Как видно по этому фрагменту кода, чтобы управлять файлами в системном окружении компьютера, мы можем просто запросить стандартную файловую систему — такой подход используется в классическом API-интерфейсе. Однако класс `FileSystems` может создавать объекты `FileSystem` на основе специального идентификатора URI. Позже в этой главе мы покажем пример работы с файловой системой, основанной на ZIP-архивах (но сначала нам нужно будет обсудить сжатие данных).

Класс `FileSystem` реализует интерфейс `Closeable`, поэтому вызов метода `close()` приводит к закрытию всех его файловых каналов и других потоковых объектов. В этот момент любая попытка чтения или записи в таких каналах будет приводить к генерированию исключения. Обратите внимание, что стандартную файловую систему (ту, которая используется на текущем компьютере) закрыть нельзя.

Получив объект `FileSystem`, мы можем воспользоваться его фабричным методом для создания объектов `Path`, которые представляют файлы и директории. Экземпляр `Path` может быть создан на основе строкового представления (так же, как классический объект `File`), после чего его можно использовать в сочетании с методами из класса `Files`, чтобы выполнять операции создания, чтения, записи или удаления.

```
Path fooPath = fs.getPath( "/tmp/foo.txt" );
OutputStream out = Files.newOutputStream( fooPath );
```

Этот код открывает поток `OutputStream`, чтобы записать данные в файл *foo.txt*. По умолчанию, если файл не существует, он создается автоматически. В противном случае перед началом записи новых данных он будет обнулен (его длина станет равняться нолю). Такое поведение можно изменить при помощи различных параметров. Мы поговорим о методах класса `Files` в следующем разделе.

Объект `Path` реализует интерфейс `java.lang.Iterable`, с помощью которого можно перебирать составные части пути (например, разделенные слешем имена «tmp» и «foo.txt», которые мы использовали в предыдущем примере). Но если вас больше интересует поиск других файлов и директорий по заданному пути, вам стоит обратить внимание на компоненты `DirectoryStream` и `FileVisitor`, о которых мы поговорим позже. Объект `Path` также реализует интерфейс `java.nio.file.Watchable`, который позволяет следить за изменениями в файловой иерархии (об этом пойдет речь в следующем разделе).

У класса `Path` есть вспомогательный метод для получения пути относительно файла или директории:

```
Path patPath = fs.getPath( "/User/pat/" );
Path patTmp = patPath.resolve("tmp" ); // "/User/pat/tmp"
// То же, что и выше, только с передачей объекта Path
Path tmpPath = fs.getPath( "tmp" );
Path patTmp = patPath.resolve( tmpPath ); // "/User/pat/tmp"
// Каким бы ни было значение Path, полный путь всегда
// возвращается в неизменном виде
Path absPath = patPath.resolve( "/tmp" ); // "/tmp"
// Получаем соседний элемент по отношению к patPath
// (в той же директории)
Path danPath = patPath.resolveSibling( "dan" ); // "/Users/dan"
```

В этом фрагменте кода применяются методы `Pathresolve()` и `resolveSibling()`, с помощью которых мы ищем файлы и директории относительно заданного объекта `Path`. Метод `resolve()` обычно используется для добавления относительного пути к директории, представленной имеющимся объектом `Path`. Абсолютный путь, переданный в метод `resolve()`, возвращается без изменений (это похоже на работу команды «cd» в операционных системах DOS и Unix). Метод `resolveSibling()` работает схожим образом, но в качестве отправной точки он выбирает родительскую директорию заданного объекта `Path`; с помощью этого метода можно описать конечный результат операции `move()`.

Преобразование объекта Path в классический тип File и наоборот

Чтобы связать между собой старый и новый API-интерфейсы, и позволить конвертировать один вид пути в другой, в классах `java.io.File` и `java.nio.file.Path` были предусмотрены методы `toPath()` и `toFile()` соответственно. Естественно, объекты `Path`, полученные из путей типа `File`, могут представлять только те файлы и директории, которые находятся на стандартной файловой системе компьютера.

```
Path tmpPath = fs.getPath( "/tmp" );
File file = tmpPath.toFile();
File tmpFile = new File( "/tmp" );
Path path = tmpFile.toPath();
```

Файловые операции, предоставляемые пакетом NIO

Имея в своем распоряжении объект `Path`, мы можем работать с ним при помощи статических методов класса `Files`, создавая путь к файлу или директории, считывая и записывая данные по этому пути, запрашивая и устанавливая свойства элементов файловой системы. В этом разделе мы перечислим множество таких методов, останавливаясь на самых важных из них. В следующем таблице представлен список методов класса `java.nio.file.Files`. Это довольно длинный список, что неудивительно, ведь класс `Files` выполняет все виды операций с файлами. Чтобы сделать таблицу более легкой для восприятия, мы опустили перегруженные версии одних и тех же методов (те, что принимают разные наборы аргументов), сгруппировав похожие и связанные между собой операции.

Табл. 12.2. Методы класса `Files` из пакета NIO

Метод	Возвращаемый тип	Описание
`copy()`	`long` или `Path`	Копирует поток в файл по заданному пути, файл по заданному пути в поток или один путь в другой. Возвращает количество скопированных байтов или целевой путь. Если целевой файл уже существует, его можно заменить (хотя по умолчанию операция завершается неудачей). Результатом копирования директории будет пустая директория по заданному пути (содержимое не копируется). При копировании символьных ссылок копируется содержимое файлов, на которые они указывают (что выливается в обычное копирование файлов).

Метод	Возвращаемый тип	Описание
`createDirectory()`, `createDirectories()`	`Path`	Создают одну или все директории по указанному пути. Если директория уже существует, метод `createDirectory()` сгенерирует исключение, а метод `createDirectories()` проигнорирует имеющиеся директории и создаст только те, которых еще нет.
`createFile()`	`Path`	Создает пустой файл. Это атомарная операция, она выполнится успешно, только если файла не существует. (С помощью этого свойства можно создавать файлы для идентификации и защиты ресурсов.)
`createTempDirectory()`, `createTempFile()`	`Path`	Создают временные, гарантированные файлы и директории с уникальными именами и указанным префиксом. Опционально могут размещать их в стандартной временной директории системы.
`delete()`, `deleteIfExists()`	`void`	Удаляют файл или пустую директорию. Метод `deleteIfExists()` не генерирует исключение, если файл уже существует.
`exists()`, `notExists()`	`boolean`	Определяют, существует ли файл (метод `notExists()` просто возвращает противоположное значение). Опционально можно указать, нужно ли переходить по ссылкам (по умолчанию переход выполняется).
`exists()`, `isDirectory()`, `isExecutable()`, `isHidden()`, `isReadable()`, `isRegularFile()`, `isWriteable()`	`boolean`	Проверяют основные свойства файла: существует ли указанный путь, является ли этот элемент директорией и т. д.
`createLink()`, `createSymbolicLink()`, `isSymbolicLink()`, `readSymbolicLink()`, `createLink()`	`boolean` или `Path`	Создают жесткую или символьную ссылку, проверяют, является ли файл символьной ссылкой, считывают файл, находящийся по символьной ссылке. Символьные ссылки — это всего лишь файлы, которые указывают на другие файлы. Обычные («жесткие») ссылки являются низкоуровневыми псевдонимами, когда два имени могут указывать на одни и те же исходные данные. Если вы не уверены, какой тип ссылок лучше выбрать, используйте символьные.
`getAttribute()`, `setAttribute()`, `getFileAttributeView()`, `readAttributes()`	`Object`, `Map` или `FileAttributeView`	Получают или устанавливают атрибуты, связанные с файловой системой. Например, время доступа и изменения, подробные права доступа, а также информация о владельце. Имена зависят от реализации.

Метод	Возвращаемый тип	Описание
`getFileStore()`	`FileStore`	Получает объект `FileStore`, который представляет устройство, том или другой вид раздела файловой системы, на который указывает путь.
`getLastModifiedTime()`, `setLastModifiedTime()`	`FileTime` или `Path`	Получают или устанавливают время последнего изменения файла или директории.
`getOwner()`, `setOwner()`	`UserPrincipal`	Получают или устанавливают объект `UserPrincipal`, который представляет владельца файла. Чтобы имя пользователя в виде строки, применяйте методы `toString()` или `getName()`.
`getPosixFile Permissions()`, `setPosixFile Permissions()`	`Set` или `Path`	Получают или устанавливают полное описание прав доступа на чтение и запись в стиле POSIX (пользователь-группа-другие). Для этого используется набор (`Set`) перечислений типа `PosixFilePermission`.
`isSameFile()`	`boolean`	Проверяет, указывают ли два пути на один и тот же файл (может вернуть `true`, даже если пути не идентичны).
`move()`	`Path`	Перемещает файл или директорию путем переименования или копирования. Можно указать, нужно ли заменять существующие элементы. Если для перемещения файла по файловой системе (или хранилищу) не требуется копирование, будет использовано переименование. Для директорий такой вариант возможен только в случае простого переименования или если директория пустая. Если для перемещения директории нужно копировать ее файлы, генерируется исключение `IOException`. (Тогда вы должны копировать файлы самостоятельно. См. `walkFileTree()`.)
`newBufferedReader()`, `newBufferedWriter()`	`BufferedReader` или `BufferedWriter`	Открывают файл для чтения посредством объекта `BufferedReader` или создают и открывают файл для записи при помощи `BufferedWriter`. В обоих случаях нужно указывать кодировку символов.
`newByteChannel()`	`SeekableByteChannel`	Создает новый файл или открывает существующий в виде байтового канала с возможностью перехода. (Более подробно об этом рассказывается позже в этой главе, в разделе о пакете NIO.) В качестве альтернативы можно использовать метод `FileChannel.open()`.
`newDirectoryStream()`	`DirectoryStream`	Возвращает объект DirectoryStream для перебора иерархии директорий. Можно указать фильтр или шаблон для имен файлов.

Метод	Возвращаемый тип	Описание
`newInputStream()`, `newOutputStream()`	`InputStream` или `OutputStream`	Открывают файл для чтения посредством объекта `InputStream` или создают и открывают файл для записи при помощи `OutputStream`. Можно сделать так, чтобы для исходящего потока файлы обнулялись; по умолчанию обнуление происходит во время записи.
`probeContentType()`	`String`	Возвращает MIME-тип файла, если его можно определить с помощью установленных сервисов класса `FileTypeDetector`. В противном случае возвращается `null`.
`readAllBytes()`, `readAllLines()`	`byte[]` или `List<String>`	Считывают все данные из файла в виде массива байтов (`byte []`) или все символы в виде списка строк, используя указанную кодировку.
`size()`	`long`	Получает размер файла (в байтах) по заданному пути.
`walkFileTree()`	`Path`	Применяет объект `FileVisitor` к заданной иерархии директорий. С помощью аргументов определяются максимальная глубина перебора и необходимость перехода по ссылкам.
`write()`	`Path`	Записывает в файл по заданному пути массив байтов или коллекцию строк (с указанной кодировкой), после чего закрывает его. При желании можно задать поведение относительно добавления данных и обнуления файла. По умолчанию файл обнуляется, и его содержимое перезаписывается.

С помощью вышеописанных методов мы можем получать для заданного файла входящие и исходящие потоки или буферизированные объекты `Reader` и `Writer`. Мы также можем создавать пути к файлам и директориям, перебирать иерархии файловой системы. Операции с директориями мы обсудим в следующем разделе.

Напомним, что с помощью методов `resolve()` и `resolveSibling()` из класса `Path` можно создавать параметры для операций `copy()` и `move()`.

```
// Перемещаем файл /tmp/foo.txt в /tmp/bar.txt
Path foo = fs.getPath("/tmp/foo.txt" );
Files.move( foo, foo.resolveSibling("bar.txt") );
```

Чтобы быстро считывать и записывать содержимое файлов, не прибегая к использованию потоков, можно задействовать методы

readAllLines/readAllBytes и write, которые способны перемещать массивы строк и байтов из и внутрь файлов в рамках одной операции. Они отлично подходят для работы с файлами, которые легко помещаются в оперативную память.

```
// Считываем и записываем коллекцию
// строк (например, строчек текста)
Charset asciiCharset = Charset.forName("US-ASCII");
List<String> csvData = Files.readAllLines( csvPath, asciiCharset );
Files.write( newCSVPath, csvData, asciiCharset );
// Считываем и записываем байты
byte [] data = Files.readAllBytes( dataPath );
Files.write( newDataPath, data );
```

Операции с директориями

Помимо основных операций создания и управления директориями из класса Files существуют методы для перечисления файлов внутри заданной директории и перебора всех элементов внутри файловой иерархии. Чтобы получить список файлов в директории, мы можем воспользоваться одной из версий метода newDirectoryStream(), которая возвращает объект DirectoryStream (наследованный от интерфейса Iterable).

```
// Выводим файлы и директории, хранящиеся внутри /tmp
try ( DirectoryStream<Path> paths = Files.newDirectoryStream(
   fs.getPath( "/tmp" ) ) ) {
   for ( Path path : paths ) { System.out.println( path ); }
}
```

Этот код использует поток DirectoryStream для перебора всех записей в директории */tmp*, попутно выводя результат на экран. Стоит заметить, что открытие потока выполняется внутри конструкции try-with-resources, поэтому закрытие происходит автоматически. Как и любой другой поток, объект DirectoryStream предназначен для однонаправленного перебора, поэтому он должен быть закрыт для освобождения связанных с ним ресурсов. API-интерфейс не определяет порядок, в котором возвращаются записи, поэтому вам, возможно, придется сохранять все элементы для дальнейшей сортировки, если того требует задача.

Еще одна версия метода newDirectoryStream() принимает *шаблоны подстановки*, чтобы сузить набор перечисляемых файлов:

```
// Файлы в директории /tmp, которые
// соответствуют шаблону "*.txt"
try ( DirectoryStream<Path> paths = Files.newDirectoryStream(
   fs.getPath( "/tmp" ), "*.txt" ) ) {
   ...
```

Таким образом можно фильтровать имена файлов, используя уже знакомый нам символ * и некоторые другие шаблоны подстановки. В табл. 12.3 приведено несколько примеров таких шаблонов.

Табл. 12.3. Примеры шаблонов подстановки

Шаблон	Пример
`*.txt`	Имена файлов, заканчивающиеся на «.txt»
`*.{java,class}`	Имена файлов, заканчивающиеся на «java» или «class»
`[a,b,c]*`	Имена файлов, начинающиеся с «a», «b» или «c»
`[0-9]*`	Имена файлов, начинающиеся цифрами от 0 до 9
`[!0-9]*`	Имена файлов, начинающиеся с любых символов кроме цифр от 0 до 9
`pass?.dat`	Имена файлов, начинающиеся с «pass» плюс любой символ плюс «dat» (например, «pass1.dat», «passN.dat»)

Если шаблоны подстановки вам не подходят, вы можете предоставить собственный потоковый фильтр, реализовав интерфейс `DirectoryStream.Filter`. Ниже мы воплотили в коде шаблон «*.txt», который ищет имена файлов, заканчивающиеся на «.txt». Чтобы сделать пример короче, мы реализовали наш фильтр в виде внутреннего анонимного класса:

```
// То же, что и выше, только с использованием нашей
// собственной (анонимной) реализации фильтра
try ( DirectoryStream<Path> paths = Files.newDirectoryStream(
   fs.getPath( "/tmp" ),
   new DirectoryStream.Filter<Path>() {
      @Override
      public boolean accept( Path entry ) throws IOException {
         return entry.toString().endsWith( ".txt" );
      }
} ) ) {
   ...
```

И, наконец, если нам необходимо пройтись не по одной директории, а по всей ее иерархии, мы можем воспользоваться интерфей-

сом FileVisitor. Метод FileswalkFileTree() принимает начальный путь и выполняет перебор дерева каталогов, предоставляя объекту FileVisitor возможность «посетить» каждый элемент иерархии. Небольшой фрагмент кода, представленный ниже, выводит на экран все файлы и директории, которые находятся по пути **/Users/pat**:

```
// Посещаем все файлы в иерархии директорий
Files.walkFileTree( fs.getPath( "/Users/pat"), new
SimpleFileVisitor<Path>() {
   @Override
   public FileVisitResult visitFile( Path file, BasicFileAttributes attrs
)
   {
      System.out.println( "path = " + file );
      return FileVisitResult.CONTINUE;
   }
});
```

Метод visitFile() из нашего объекта FileVisitor вызывается для каждого элемента иерархии файлов; в качестве аргументов ему передаются объект Path и атрибуты файла. Объект FileVisitor может делать с файлом все что угодно, возвращая перечисления типа FileVisitResult.CONTINUE или FileVisitResult.TERMINATE, чтобы сигнализировать о том, стоит ли дальше продолжать обход дерева.

Здесь для удобства используется класс SimpleFileVisitor, который содержит «холостые» (пустые) реализации методов интерфейса FileVisitor, благодаря чему мы можем переопределять только интересующие нас операции. Нам доступны и другие методы: например, visitFileFailed() вызывается, когда файл или директорию нельзя посетить (возможно, из-за прав доступа), а парные операции preVisitDirectory() и postVisitDirectory() позволяют выполнять действия до и после посещения новой директории. Метод preVisitDirectory() может возвращать два полезных значения, SKIP_SUBTREE и SKIP_SIBLINGS; первое позволяет пропустить заданную ветвь иерархии, не прерывая весь процесс, а второе — продолжать перебор без углубления в соседние директории того же уровня.

Как видите, для перечисления файлов и обхода иерархии директорий в пакете NIO предусмотрены куда более мощные методы, которые могут послужить прекрасным дополнением к классическому API-интерфейсу java.io.

Отслеживание путей

Одним из наиболее полезных компонентов нового пакета `java.nio.file` является интерфейс `WatchService`, который позволяет следить за изменениями в файлах и директориях по заданному пути `Path`. Мы можем получать события, связанные с созданием, изменением и удалением элементов файловой системы. Следующий фрагмент кода следит за изменениями в каталоге */Users/pat*:

```
Path watchPath = fs.getPath("/Users/pat");
WatchService watchService = fs.newWatchService();
watchPath.register( watchService, ENTRY_CREATE, ENTRY_MODIFY, ENTRY_DELETE
);
while( true )
{
   WatchKey changeKey = watchService.take();
   List<WatchEvent<?>> watchEvents = changeKey.pollEvents();
   for ( WatchEvent<?> watchEvent : watchEvents )
   {
      // Все наши события имеют тип Path:
      WatchEvent<Path> pathEvent = (WatchEvent<Path>)watchEvent;
      Path path = pathEvent.context();
      WatchEvent.Kind<Path> eventKind = pathEvent.kind();
      System.out.println( eventKind + " for path: " + path );
   }
   changeKey.reset(); // Важно!
}
```

С помощью метода `newWatchService()` из объекта `FileSystem` мы создаем сервис `WatchService`. Далее мы можем зарегистрировать для этого сервиса реализацию интерфейса `Watchable` (пока что в этом качестве выступает объект `Path`), события которого будут отслеживаться. Как видно, в API-интерфейсе все устроено наоборот: метод `register()` вызывается именно из объекта `Watchable`, а аргументами для него служат сам сервис и список параметров переменной длины, которые представляют интересующие нас типы событий (`ENTRY_CREATE`, `ENTRY_MODIFY` или `ENTRY_DELETE`). Есть еще один дополнительный тип, `OVERFLOW`; события, которые можно получать с его помощью, сигнализируют о том, что механизм мониторинга файловой системы слишком медленный для того, чтобы обработать все изменения, поэтому часть событий может до нас не дойти.

Закончив с приготовлениями, мы можем начать запрашивать информацию об изменениях с помощью метода `take()`, который возвращает объект `WatchKey`. Метод `take()` блокирует поток выполнения до тех пор, пока не появится событие; существует также неблокирующая версия этой операции под названием `poll()`. Мы можем извлечь события из объекта `WatchKey`, воспользовавшись методом `pollEvents()`. Здесь мы снова сталкиваемся с некоторой неочевидностью API-интерфейса, так как `WatchEvent` — это обобщенный тип, параметризированный для работы с объектом `Watchable`. В нашем случае в качестве событий всегда выступают объекты `Path`, которые при необходимости можно привести к нужному нам типу. Вид события (создание, изменение, удаление) можно узнать при помощи метода `WatchEventkind()`, а метод `context()` возвращает измененный путь. В конце важно не забыть вызвать метод `reset()` из объекта `WatchKey`, чтобы очистить список событий и иметь возможность дальше получать информацию об изменениях.

Производительность сервиса `WatchService` во многом зависит от его реализации. Часто механизм мониторинга файловой системы встроен в ОС, и это позволяет получать события практически мгновенно. Но нередко платформа Java полагается на собственную универсальную реализацию, основанную на фоновых потоках, которая очень долго обнаруживает изменения. Например, на момент написания этой книги версия платформы Java 7 для OS X не использует преимущества системного мониторинга файлов, получая данные из медленного универсального сервиса.

Сериализация

С помощью потока `DataOutputStream` вы можете написать приложение, которое последовательно преобразует содержимое ваших объектов в значения простых типов. Однако платформа Java предоставляет еще более мощный механизм, который способен выполнить за вас всю работу. Речь идет о *сериализации* — автоматическом способе сохранения и загрузки состояния объекта. Это очень обширная тема, которую мы не в состоянии полностью охватить в рамках данной книги. К примеру, вы не найдете здесь некоторые интересные приемы вроде ведения версий, равно как и сведения об управлении процессом сериализации.

В сущности, экземпляр любого класса, который реализует интерфейс `Serializable`, может быть сохранен в поток и восстановлен обратно. Для сериализации простых типов и объектов используются разновидности

потоков `ObjectInputStream` и `ObjectOutputStream`. Производные классов, реализующих интерфейс `Serializable`, тоже могут быть сериализованы. Стандартный механизм сериализации сохраняет значения всех полей объекта (публичных и частных), за исключением статических и тех, которые помечены ключевым словом `transient`.

Одна из наиболее важных (и неочевидных) особенностей данного механизма заключается в том, что вместе с объектом сериализуются все ссылки, которые он содержит. Этот процесс может охватить весь «граф» взаимосвязанных объектов, и во время восстановления все эти связи остаются на месте (мы продемонстрируем это в следующем примере). Смысл в том, что любой сериализуемый нами объект должен содержать ссылки только на те объекты, которые реализуют интерфейс `Serializable`. Мы можем отсечь часть иерархии и ограничить содержимое объекта, которое должно сериализоваться, помечая переменные ключевым словом `transient` или переопределяя стандартный механизм сериализации. Модификатор `transient` можно применить к любому полю, и тем самым указать, что содержимое этого поля не имеет ценности за пределами текущего контекста, и поэтому не должно сохраняться.

В следующем примере мы создаем хеш-таблицу и сохраняем ее содержимое в файл под названием *hash.ser*. Объект `Hashtable` изначально поддается сериализации, так как он реализует интерфейс `Serializable`.

```
import java.io.*;
import java.util.*;
public class Save {
   public static void main(String[] args) {
      Hashtable hash = new Hashtable();
      hash.put("string", "Габриэль Гарсиа Маркес");
      hash.put("int", new Integer(26));
      hash.put("double", new Double(Math.PI));
      try {
         FileOutputStream fileOut = new FileOutputStream( "hash.ser" );
         ObjectOutputStream out = new ObjectOutputStream( fileOut );
         out.writeObject( hash );
         out.close();
      }
      catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

Сначала мы создаем хеш-таблицу с несколькими элементами внутри. Затем, в рамках блока `try`, мы записываем содержимое этой хеш-таблицы в файл под названием *hash.ser*, используя метод `writeObject()` из объекта `ObjectOutputStream`. Главное различие между классами `ObjectOutputStream` и `DataOutputStream` как раз и заключается в наличии этого мощного метода.

Созданная нами хеш-таблица имеет внутренние ссылки на элементы, которые она содержит. Таким образом, элементы автоматически сериализуются вместе с таблицей. В этом можно убедиться на примере следующего кода, который восстанавливает (десериализует) объект `Hashtable`.

```
import java.io.*;
import java.util.*;
public class Load {
   public static void main(String[] args) {
      try {
         FileInputStream fileIn = new FileInputStream("hash.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         Hashtable hash = (Hashtable)in.readObject();
         System.out.println( hash.toString() );
      }
      catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

В этом примере мы считываем хеш-таблицу из файла **hash.ser**, используя метод `readObject()` из объекта `ObjectInputStream`. В результате мы получаем экземпляр `Object`, который нужно привести к типу `Hashtable`. В самом конце мы выводим содержимое хеш-таблицы, вызывая из нее метод `toString()`.

Инициализация объектов с помощью метода `readObject()`

Часто простой десериализации бывает вполне достаточно, чтобы полностью восстановить состояние объекта. К примеру, некоторые поля объекта, состояние которых нельзя сериализовать (это может быть се-

тевое подключение, регистрация событий или декодированное изображение), можно пометить ключевым словом `transient`. Однако у вас есть возможность самостоятельно подготовить объект к работе после его десериализации; для этого необходимо переопределить специальный метод под названием `readObject()`.

В отличие от одноименного метода из класса `ObjectInputStream`, эта версия `readObject()` реализуется самим сериализуемым объектом; чтобы ее было легко распознать и использовать, она имеет специфичную сигнатуру и должна быть частной. Следующий фрагмент кода заимствован из компонента JavaBean, который отвечает за анимацию (о нем мы поговорим в главе 22):

```
private void readObject(ObjectInputStream s)
   throws IOException, ClassNotFoundException
{
   s.defaultReadObject();
   initialize();
   if ( isRunning )
      start();
}
```

Если метод с данной сигнатурой присутствует в объекте, он вызывается во время процесса восстановления.

Поток `ObjectInputStream`, который выступает в качестве аргумента, занимается созданием объекта. Сначала мы делегируем вызов методу `defaultReadObject()`, чтобы выполнить обычную десериализацию из потока, а затем сами приступаем к подготовительным процедурам.

В данном случае мы вызываем один из наших методов под названием `initialize()`, а потом, если того требует состояние объекта, запускаем метод `start()`.

С помощью собственной реализации `readObject()` и соответствующего метода `writeObject()` можно полностью контролировать сериализованную версию объекта; вместо делегирования вызова стандартной реализации, как было показано выше, мы могли бы считывать и записывать поток данных с применением низкоуровневых операций (байтовых, строковых и т. д.).

Мы еще вернемся к теме сериализации в главе 22, когда речь пойдет о технологии JavaBeans.

Идентификатор `serialVersionUID`

Сериализация объектов в платформе Java была спроектирована с учетом возможных *изменений* в структуре классов, которые не приводят к *несовместимости*. Например, изменение методов класса не обязательно должно приводить к изменению сериализованной версии объекта, так как сохраняется только содержимое переменных. Добавление в класс нового поля тоже не должно помешать загрузке старой сериализованной версии объекта. Мы просто можем позволить новой переменной принимать какое-то стандартное значение. Однако по умолчанию платформа Java ведет себя довольно скрупулезно и осторожно. Если вы хоть немного измените структуру своего класса, и попытаетесь прочитать его предыдущую сериализованную версию, будет сгенерировано исключение `InvalidClassException`.

Эти версии платформа Java определяет с помощью функции хеширования, которая применяется к структуре класса. Результат выполнения этой функции хранится в виде 64-битного значения вместе с сериализованными данными и называется *уникальным идентификатором сериализованной версии* (Serial Version UID или SUID). Вы можете сравнивать его с хешем загружаемого вами класса.

Платформа Java позволяет контролировать этот процесс с помощью специального «магического» поля, которое присутствует в нашем классе и выглядит следующим образом:

```
static final long serialVersionUID = -6849794470754667710L;
```

(Конечно же, каждый класс имеет свое уникальное значение.) Если платформа Java находит внутри класса статическое поле `serialVersionUID` типа `long`, она использует его вместо результата выполнения хеш-функции. Это значение записывается во время сериализации объектов и используется для сравнения их восстановленных версий. То есть теперь мы сами можем определять, какие версии класса совместимы с сериализованным объектом. Например, мы можем с самого начала снабдить наш класс идентификатором SUID, инкрементируя его значения только в том случае, если вносимые нами изменения приводят к несовместимости; это позволит предотвратить загрузку старых версий класса:

```
class MyDataObject implements Serializable {
   static final long serialVersionUID = 1; // Версия 1
   ...
}
```

Утилита командной строки *serialver*, которая входит в состав пакета JDK, позволяет вычислять хеш класса по тому же алгоритму, который использует платформа Java. Она незаменима в тех случаях, когда у вас уже имеются сериализованные объекты и вам необходимо изменить их класс. Команда *serialver* выводит для заданного класса идентификатор SUID, который гарантированно совпадает с ранее сохраненным значением:

```
% serialver SomeObject
static final long serialVersionUID = -6849794470754667710L;
```

Поместив это значение в свой класс, вы сможете «заморозить» идентификатор SUID; это позволит изменять код класса, не влияя на версию сериализуемых объектов.

Сжатие данных

Пакет `java.util.zip` содержит классы, с помощью которых можно сжимать данные в потоках и файлах. Поддерживается два широко распространенных формата: GZIP и ZIP. В этом разделе мы поговорим о применении данных классов. Мы также покажем вам два демонстрационных приложения, основанных на компонентах, которые мы изучили в этой главе. В конце мы обсудим высокоуровневые инструменты, появившиеся в Java 7, которые позволяют работать с ZIP-архивами как с файловой системой.

Архивы и сжатие данных

Пакет `java.util.zip` предоставляет два фильтрующих потока для записи сжатых данных. Класс `GZIPOutputStream` предназначен для сохранения информации в формате GZIP. Его аналог, `ZIPOutputStream`, занимается созданием сжатых ZIP-архивов, которые могут содержать любое количество файлов. Чтобы записать сжатые данные в формате GZIP, используйте объект `GZIPOutputStream` в качестве обертки для исходного потока. В следующем примере представлена полноценная программа, которая демонстрирует процесс сжатия файла в формат GZIP, хотя с тем же успехом мы могли бы получать наши данные из сетевого соединения или любого другого потока. Приложение GZip является утилитой командной строки и предназначено для сжатия файлов.

```
import java.io.*;
import java.util.zip.*;
public class GZip {
public static int sChunk = 8192;
public static void main(String[] args) {
if (args.length != 1) {
System.out.println("Usage: GZip source");
return;
}
// создаем исходящий поток
String zipname = args[0] + ".gz";
GZIPOutputStream zipout;
try {
FileOutputStream out = new FileOutputStream(zipname);
zipout = new GZIPOutputStream(out);
}
catch (IOException e) {
System.out.println("Couldn't create " + zipname + ".");
return;
}
byte[] buffer = new byte[sChunk];
// сжимаем файл
try {
FileInputStream in = new FileInputStream(args[0]);
int length;
while ((length = in.read(buffer, 0, sChunk)) != -1)
zipout.write(buffer, 0, length);
in.close();
}
catch (IOException e) {
System.out.println("Не удается сжать " + args[0] + ".");
}
try { zipout.close(); }
catch (IOException e) {}
}
}
```

Сначала мы убеждаемся в том, что в командной строке указан аргумент с именем файла. Затем мы добавляем к имени файла суффикс *.gz* и используем полученный результат при создании потока `FileOutputStream`, оберткой для которого служит объект `GZIPOutputStream`. Проделав все

это, мы открываем исходный файл, считываем данные небольшими порциями и записываем их в поток `GZIPOutputStream`. В конце мы убираем за собой, закрывая все наши потоки.

ZIP-архивы

В то время как GZIP — это простой формат сжатия потоковых и файловых данных, ZIP-архив представляет собой набор файлов, часть которых (или все сразу) можно сжимать. Процесс записи данных в файл ZIP-архива выглядит немного сложнее, чем просто создание обертки для потока, но это не должно вызывать никаких затруднений. Каждый элемент ZIP-файла представлен объектом `ZipEntry`. Перед тем, как записывать данные в любой элемент потока `ZipOutputStream`, нужно вызвать метод `putNextEntry()`. В следующем примере показан процесс создания объекта `ZipOutputStream`. Вы увидите, что здесь, как и при создании экземпляра `GZIPOutputStream`, все начинается с обертки вокруг потока:

```
ZipOutputStream zipout;
try {
   FileOutputStream out = new FileOutputStream("archive.zip");
   zipout = new ZipOutputStream(out);
}
catch (IOException e) {}
```

Допустим, у нас есть два файла, которые мы хотим поместить в этот архив. Прежде чем начинать запись, нам нужно вызвать метод `putNextEntry()`, чтобы задать имя файла внутри архива и указать соответствующую позицию в потоке. Здесь мы создаем простой вариант элемента `ZipEntry`, который содержит только имя файла. Мы могли бы указать и другие поля, связанные с форматом ZIP, но в большинстве случаев об этом можно не беспокоиться.

```
try {
   ZipEntry entry = new ZipEntry("first.dat");
   zipout.putNextEntry(entry);
   zipout.write( ... ) // Записываем данные для первого файла
   ZipEntry entry = new ZipEntry("second.dat");
   zipout.putNextEntry(entry);
   zipout.write( ... ) // Записываем данные для второго файла
   . . .
   zipout.close();
}
catch (IOException e) {}
```

Распаковка данных

Чтобы распаковать данные, сжатые в формате GZIP, нужно просто создать вокруг исходного потока `FileInputStream` обертку типа `GZIPInputStream` и считывать информацию из нее. Следующий пример дополняет программу GZip, созданную нами ранее, демонстрируя процесс распаковки GZIP-файла:

```
import java.io.*;
import java.util.zip.*;
public class GUnzip {
public static int sChunk = 8192;
public static void main(String[] args) {
if (args.length != 1) {
System.out.println("Использование: GUnzip источник");
return;
}
// создаем входящий поток
String zipname, source;
if (args[0].endsWith(".gz")) {
zipname = args[0];
source = args[0].substring(0, args[0].length() - 3);
}
else {
zipname = args[0] + ".gz";
source = args[0];
}
GZIPInputStream zipin;
try {
FileInputStream in = new FileInputStream(zipname);
zipin = new GZIPInputStream(in);
}
catch (IOException e) {
System.out.println("Не удается открыть " + zipname + ".");
return;
}
byte[] buffer = new byte[sChunk];
// распаковываем файл
try {
FileOutputStream out = new FileOutputStream(source);
int length;
```

```
while ((length = zipin.read(buffer, 0, sChunk)) != -1)
out.write(buffer, 0, length);
out.close();
}
catch (IOException e) {
System.out.println("Не удается распаковать " + args[0] + ".");
}
try { zipin.close(); }
catch (IOException e) {}
}
}
```

Сначала мы убеждаемся в том, что в командной строке указан аргумент с именем файла. Если аргумент заканчивается на «.gz», мы вычисляем, каким должно быть имя распакованного файла. В противном случае мы используем заданный аргумент исходя из того, что сжатый файл имеет суффикс «.gz». Затем мы создаем обертку типа `GZIPInputStream` для потока `FileOutputStream`, которая будет представлять наш сжатый файл. Сделав это, мы открываем итоговый файл и поэтапно записываем в него данные, которые считываются из объекта `GZIPInputStream`. В конце мы убираем за собой, закрывая все наши потоки. Операции чтения и записи ZIP-архивов тоже являются зеркальными. Перед чтением каждого элемента из потока `ZipInputStream` нужно вызывать метод `getNextEntry()`. Если все элементы уже прочитаны, этот метод возвращает `null`. В следующем примере показан процесс создания потока `ZipInputStream`:

```
ZipInputStream zipin;
try {
   FileInputStream in = new FileInputStream("archive.zip");
   zipin = new ZipInputStream(in);
}
catch (IOException e) {}
```

Предположим, нам нужно прочитать из этого архива два файла. Прежде чем приступать к чтению, следует вызвать метод `getNextEntry()`. Каждая запись содержит как минимум имя элемента, который мы считываем из архива:

```
try {
   ZipEntry first = zipin.getNextEntry();
   zipin.read( ... ) // Считываем данные файла
} catch (IOException e) {}
```

Теперь можно прочитать содержимое первого элемента в архиве. В конце метод `read()` вернет –1; это будет означать, что вы опять можете сделать вызов `getNextEntry()`, чтобы перейти к чтению следующего элемента. Когда все элементы в архиве будут прочитаны, метод `getNextEntry()` вернет `null`.

ZIP-архив в качестве файловой системы

Одно из преимуществ нового пакета `java.nio.file` заключается в возможности создания собственных файловых систем прямо на языке Java. (Ранее в этой главе мы уже обсуждали API-интерфейс для работы с файлами из пакета NIO, и в следующем разделе мы еще вернемся к инструментам более широкого назначения, которые предлагает эта технология.) Одна из таких реализаций входит в состав платформы Java 7 и называется Zip Filesystem Provider (ZFP)*. Она позволяет открывать ZIP-архивы и обращаться с их содержимым как с файловой системой; мы можем читать, записывать, копировать и переименовывать файлы, используя стандартный API-интерфейс из пакета `java.nio.file`. Нужно только помнить, что все эти операции происходят внутри ZIP-архива, а не на файловой системе компьютера (как можно было бы ожидать).

Это стало возможным благодаря тому, что в основе API-интерфейса для работы с файлами из состава пакета NIO лежит абстрактный класс `FileSystem`, который служит фабрикой для объектов `Path`. Ранее при работе с пакетом `java.nio.file` мы всегда запрашивали стандартную файловую систему, используя метод `Filesystems.getDefault()`. На этот раз мы выберем конкретный вид файловой системы и ее местоположение, создав специальный идентификатор URI для нашего ZIP-архива. (Данный идентификатор является своего рода более абстрактной версией URL-адреса, о чем мы еще поговорим в главах, посвященных работе с сетью.)

```
// Формируем идентификатор URI, который указывает на ZIP-архив
URI zipURI = URI.create("jar:file:/Users/pat/tmp/MyArchive.zip");
// Открываем или создаем архив и записываем в него файл
Map<String, String> env = new HashMap<>();
env.put("create", "true");
try ( FileSystem zipfs = FileSystems.newFileSystem( zipURI, env ) )
```

* Файловая система ZFP поставляется в качестве примера вместе с образцами исходного кода, хотя доподлинно неизвестно, рассматривает ли ее компания Oracle в качестве некоего стандарта. На момент написания этой книги файловая система ZFP поддерживается на всех платформах и входит в состав программных пакетов JDK и JRE версии 7.

```
{
   Path path = zipfs.getPath("/README.txt");
   OutputStream out = Files.newOutputStream( path );
   try ( PrintWriter pw = new PrintWriter(
      new OutputStreamWriter( out ) ) ) {
      pw.println("Привет, мир!");
   }
}
```

В этом фрагменте кода мы воспользовались методом `URIcreate()`, чтобы сформировать для нашего ZIP-архива идентификатор URI с префиксом «jar:file:». (Формат JAR, который используется платформой Java, на самом деле является обычным ZIP-файлом с некоторыми дополнительными условностями.) Затем мы передали этот идентификатор в метод `newFileSystem()` из класса `FileSystems`, чтобы создать ссылку на файловую систему подходящего типа. Объект `FileSystem`, который мы получим в итоге, будет вести себя привычным для нас образом, только все его операции будут касаться записей внутри ZIP-архива. В качестве еще одного аргумента для метода `newFileSystem()` выступает ассоциативный массив `Map` со строковыми свойствами внутри, которые зависят от типа ФС. В данном случае мы передаем ключ «create» со значением «true», заставляя тем самым объект `FileSystem` создать архив, если такового не существует. Чтобы узнать список всех свойств, которые могут быть переданы, вам следует свериться с документацией для интересующей вас файловой системы.

В примере, представленном выше, мы создали путь `Path` для файла */README.txt* в корневом каталоге файловой системы и затем записали туда строку. Чтобы инкапсулировать процесс открытия файловой системы и записи в файл, мы использовали конструкцию try-with-resources, поэтому после завершения этих операций ресурсы будут автоматически освобождены. Все дальнейшие операции выполняются так же, как и в случае с «обычными» файлами. Например, при помощи стандартного метода `move()` из класса `Files` мы можем переместить файл, создав путь к нему и к его новому местоположению.

```
// Перемещаем файл
try ( FileSystem zipfs = FileSystems.newFileSystem( fsURI, env ) )
{
   Path path = zipfs.getPath("/README.txt");
   Path toPath = zipfs.getPath("/README2.txt");
   Files.move( path, toPath );
}
```

Пакет NIO

В этом разделе мы опять возвращаемся к пакету `java.nio`, чтобы завершить наше знакомство с основными инструментами ввода/вывода, которые предоставляет платформа Java. Название NIO расшифровывается как «new I/O» (новый ввод/вывод), и, как мы уже могли убедиться при работе с API-интерфейсом `java.nio.file`, одним из аспектов этой технологии является обновление и улучшение возможностей старого пакета `java.io`. И хотя большая часть функциональности пакета NIO действительно пересекается с уже имеющимися API-интерфейсами, изначально он был нацелен на решение специфических проблем, связанных с масштабированием крупных систем, особенно в разрезе сетевых приложений. Следующий раздел посвящен основным элементам технологии NIO, работа которых связана с *буферами* и *каналами*.

Асинхронный ввод/вывод

Главной причиной возникновения пакета NIO послужила необходимость иметь *неблокирующий* и *выборочный* ввод/вывод. Раньше большинство операций чтения и записи на платформе Java были привязаны к потокам выполнения и могли многократно блокироваться в непредсказуемые моменты времени. Определенные API-интерфейсы, такие как сокеты (которые мы рассмотрим в главе 13), предоставляли специальные средства для ограничения продолжительности выполнения операций ввода/вывода, однако это было всего лишь временное решение, которое компенсировало нехватку более универсального механизма. Во многих языках (даже в тех, которые не поддерживают многопоточность) эффективная работа операций ввода/вывода достигается за счет неблокирующего режима и возможности проверить, готовы ли эти операции отправлять или принимать данные. В неблокирующем режиме операции чтения и записи берут на себя ровно такой объем работы, который можно выполнить мгновенно; они наполняют или освобождают буфер и затем сразу возвращают результат. В сочетании с возможностью проверки готовности это позволяет однопоточным приложениям постоянно и эффективно обслуживать множество каналов. Приложение «выбирает» свободный поток выполнения и использует его до тех пор, пока тот не заблокируется, после чего переходит к следующему потоку. В однопроцессорных системах это, по сути, является аналогом многопоточности. Но, как оказывается, такой стиль обработки обеспечивает лучшее масштабирование даже в случае использования нескольких потоков (вместо одного). Подробно данный подход будет рассмотрен в главе

13, когда речь пойдет о работе с сетью и серверах для сборки приложений, способных одновременно обсуживать множество клиентов.

Помимо неблокирующего и выборочного ввода/вывода пакет NIO позволяет асинхронно закрывать и прерывать операции чтения и записи. Как уже говорилось в главе 9, до появления нового API-интерфейса на платформе Java не существовало ни единого надежного способа остановить или восстановить работу потоков выполнения, занятых вводом/выводом. Теперь же такие потоки сами «просыпаются», когда их прерывают, или когда закрывается их канал. К тому же, если вы прервете выполнение работы потока в момент, когда он заблокирован операцией из пакета NIO, его канал будет закрыт автоматически (несмотря на свою строгость, такой подход обычно себя оправдывает).

Производительность

В основе каналов для ввода/вывода лежит концепция *буфера*; буфер — это всего лишь специальный вид массива, предназначенный для передачи данных. Пакет NIO также поддерживает концепцию *прямых буферов*, которые хранят информацию за пределами Java-машины, прямо в памяти операционной системы. На практике это позволяет значительно повысить эффективность всех операций ввода/вывода. Данные могут перемещаться напрямую между двумя внешними участками памяти, избегая копирования внутрь и из Java-машины.

Блокирование файлов и отображение их в память

Технология NIO предоставляет две универсальные возможности, связанные с файлами, которые отсутствуют в пакете `java.io`. Во-первых, мы можем магическим образом отображать содержимое файла в оперативную память и работать с ним оттуда (мы поговорим об этом позже). Во-вторых, речь идет о механизме блокирования файлов, который поддерживает разделяемые и эксклюзивные блокировки отдельных частей файла — это может пригодиться при параллельном доступе из нескольких приложений.

Каналы

Если пакет `java.io` ориентирован на работу с потоками, то для технологии NIO основными компонентами являются каналы. *Канал* — это ко-

нечная точка коммуникации. На практике каналы мало чем отличаются от потоков, но их концепция выглядит более абстрактной и простой. Если потоки определяются с точки зрения операций ввода и вывода, представленных методами для чтения и записи байтов, то основной интерфейс канала не содержит никаких сведений о том, как именно происходит передача данных. Благодаря методам `isOpen()` и `close()` этот интерфейс умеет только две вещи: быть открытым и закрытым. Методы для всех остальных операций, таких как чтение, запись или передача данных, появляются на уровне конкретных реализаций каналов (для файлов, сетевых сокетов или разных устройств). В состав пакета NIO входят следующие каналы:

- `FileChannel`
- `Pipe.SinkChannel`, `Pipe.SourceChannel`
- `SocketChannel`, `ServerSocketChannel`, `DatagramChannel`

Класс `FileChannel` мы рассмотрим в этой главе. Каналы из абстрактного класса `Pipe` являются эквивалентом конвейеров из пакета `java.io`. О каналах `Socket` и `Datagram` мы поговорим в главе 13. Кроме того, с выходом платформы Java 7 появилась поддержка асинхронных версий каналов для работы с файлами и сокетами, которые, в сущности, буферизируют все свои операции через пул потоков выполнения, возвращая результаты с помощью асинхронного API-интерфейса; речь идет о классах `AsynchronousFileChannel`, `AsynchronousSocketChannel`, `AsynchronousServerSocketChannel` и `AsynchronousDatagramChannel`. Позже в этой главе мы еще вернемся к асинхронным файловым каналам.

Все эти базовые разновидности каналов реализуют интерфейс `ByteChannel`, который обеспечивает их методами для чтения и записи, похожими на те, что используются в потоках ввода/вывода. Хотя в отличие от потоков реализации `ByteChannel` считывают и записывают не обычные массивы байтов, а буферы типа `ByteBuffer`.

Кроме того, вы можете объединять вместе каналы, потоки и производные классов `Reader` и `Writer` из пакета `java.io`, чтобы обеспечивать их взаимодействие. Но нужно понимать, что, смешивая возможности разных пакетов, вы можете лишиться некоторых преимуществ технологии NIO (в том числе повышенной производительности).

Буферы

Большинство инструментов в пакетах `java.io` и `java.net` ориентировано на работу с массивами байтов. Их аналоги из пакета NIO основа-

ны на классе `ByteBuffer` (есть также символьный буфер `CharBuffer` для текста). Но у нас уже есть обычные байтовые массивы, зачем нам нужно что-то другое? Буферы решают сразу несколько задач:

- Они формализуют принцип использования буферизированных данных, реализуя концепцию буферов, доступных только для чтения. Они следят за позицией чтения/записи, учитывают ограничения при работе с большими объемами данных. Они также позволяют пометить и сбросить определенную позицию, как это делается в классе `java.io.BufferedInputStream`.
- Они предоставляют дополнительные API-интерфейсы, которые позволяют иметь дело с простыми типами, представленными в виде необработанных данных. Вы можете создавать буферы, которые «воспринимают» байты информации как набор более длинных простых значений, таких как `short`, `int` или `float`. Базовый класс для любого буфера, `ByteBuffer`, содержит методы для чтения и записи всех простых типов — в этом он похож на поток `DataOutputStream`.
- Они абстрагируют исходное хранилище данных, позволяя платформе Java выполнять специальные оптимизации. В частности, возможно выделение прямых буферов, которые вместо обычных массивов используют средства буферизации операционной системы. Благодаря интерфейсу `Channel` из пакета NIO этот процесс выполняется автоматически, как один из путей оптимизации ввода/вывода. Например, считывание файла в байтовый массив обычно подразумевает копирование данных из операционной системы в память Java-машины. Прямые буферы позволяют держать данные под контролем ОС (и за пределами внутренней памяти платформы Java) до тех пор, пока они для чего-нибудь не понадобятся.

Операции с буфером

Все буферы в пакете NIO являются производными класса `java.nio.Buffer`. Этот класс похож на массив, у которого есть состояние. Он определяет функциональность, общую для всех буферов, не уточняя тип элементов, которые могут в нем храниться (этим занимаются производные типы). Объект `Buffer` имеет фиксированный размер, который еще называют *вместимостью*. И хотя все стандартные буферы обеспечивают «произвольный доступ» к своему содержимому, объект класса `Buffer` ожидает, что данные в нем будут считываться и записываться последовательно. В связи с этим он хранит информацию о *позиции* следующего

элемента, который будет прочитан или записан. Помимо позиции состояние буфера может описываться еще двумя терминами: *граница* — позиция «ограничителя», в рамках которого может происходить чтение и запись, и *метка* — ранее отмеченная позиция, которую можно использовать в дальнейшей работе.

Реализации класса `Buffer` несут в себе специфические, типизированные методы `get()` и `put()`, которые считывают и записывают содержимое буфера. Например, класс `ByteBuffer` может читать и записывать байты и массивы байтов (у него есть много других полезных методов, которые мы обсудим позже). Операции чтения и записи сдвигают позицию метки, поэтому то, как буфер следит за своим содержимым, чем-то похоже на работу потока. Попытки чтения или записи за пределами границ метки приводят к генерированию исключений `BufferUnderflowException` или `BufferOverflowException` соответственно.

Значения метки, позиции, границы и вместимости всегда подчиняются следующей формуле:

```
метка <= позиция <= граница <= вместимость
```

Позиция для чтения и записи буфера всегда находится между меткой и границей, которые служат, соответственно, нижним и верхним пределами. Вместимость описывает физический объем буфера.

Вы можете явно указывать позицию и ограничительные метки, используя методы `position()` и `limit()`. Существует также несколько вспомогательных методов для повседневных задач. Метод `reset()` возвращает позицию обратно к метке. Если метка еще не была задана, вы получите исключение `InvalidMarkException`. Метод `clear()` сбрасывает значение позиции в 0 и отодвигает границу на уровень вместимости, подготавливая буфер к новым данным (метка исчезает). Стоит заметить, что метод `clear()` просто меняет состояние буфера, совершенно не затрагивая данные внутри.

Метод `flip()` используется для таких повседневных операций, как запись данных в буфер и чтение их оттуда. Он устанавливает границу, обнуляя текущую позицию (любая метка сбрасывается); это избавляет от необходимости следить за объемом прочитанных данных. Еще один метод, `rewind()`, просто обнуляет позицию, не затрагивая границу. С его помощью вы можете повторно записывать данные того же размера. Ниже представлен фрагмент кода, в котором данные берутся из одного канала и записываются в два других:

```
ByteBuffer buff = ...
while ( inChannel.read( buff ) > 0 ) { // позиция = ?
   buff.flip(); // граница = позиция; позиция = 0;
   outChannel.write( buff );
   buff.rewind(); // позиция = 0
   outChannel2.write( buff );
   buff.clear(); // позиция = 0; граница = вместимость
}
```

На первый взгляд, этот пример может показаться запутанным, так как чтение из канала означает запись в буфер (и наоборот). Мы записываем данные до тех пор, пока не будет достигнута граница, поэтому здесь выполнение методов `flip()` и `rewind()` приводит к одинаковым результатам.

Виды буферов

Как уже утверждалось ранее, различные виды буферов снабжают основной интерфейс методами `get()` и `put()`, которые предназначены для чтения и записи определенных типов данных. Для любого простого типа в языке Java есть соответствующий буфер: `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer` и `DoubleBuffer`. Каждый из этих классов умеет читать и записывать не только значения конкретного типа, но и массивы таких значений. Самым гибким из вышеперечисленных буферов является `ByteBuffer`. Поддерживая самые «мелкие» значения, он способен работать с любыми другими типами так же, как с типом `byte`. Ниже перечислены некоторые из его методов:

```
byte get()
char getChar()
short getShort()
int getInt()
long getLong()
float getFloat()
double getDouble()
void put(byte b)
void put(ByteBuffer src)
void put(byte[] src, int offset, int length)
void put(byte[] src)
void putChar(char value)
void putShort(short value)
void putInt(int value)
void putLong(long value)
```

```
void putFloat(float value)
void putDouble(double value)
```

Как мы уже говорили, все стандартные буферы поддерживают произвольный доступ. У каждого из вышеупомянутых методов есть альтернативная версия, которая принимает индекс, например:

```
getLong( int index )
putLong( int index, long value )
```

Но это еще не все. Из объекта `ByteBuffer` можно получить буфера любого другого типа. Например, с помощью метода `asShortBuffer()` объект `ByteBuffer` возвращает сам себя в виде экземпляра `ShortBuffer`. Этот экземпляр основан на исходном буфере — то есть он работает с теми же данными, и любые изменения будут затрагивать оба объекта. Буферы, полученные таким образом, начинаются с текущей позиции объекта `ByteBuffer`, а их вместимость определяют делением количества оставшихся байтов на размер нового типа. (Например, каждое значение типа `short` занимает два байта, `float` состоит из четырех байтов, а `long` и `double` — из восьми.) Такой подход удобен для чтения и записи больших блоков смежного типа в рамках объекта `ByteBuffer`.

Класс `CharBuffer` тоже имеет свои интересные особенности, главной из которых является интеграция со строками. Оба типа, `CharBuffer` и `String`, реализуют интерфейс `java.lang.CharSequence`, который предоставляет стандартные методы `charAt()` и `length()`. Благодаря этому в более новых API-интерфейсах (например, в пакете `java.util.regex`) типы `CharBuffer` и `String` являются взаимозаменяемыми. В этом случае объект `CharBuffer` ведет себя как строка с возможностью изменения содержимого, а также с регулируемыми начальной и конечной позициями.

Порядок следования байтов

Так как речь идет о чтении и записи типов данных, состоящих из более чем одного байта, возникает вопрос: в каком порядке записываются многобайтовые значения (такие как `short` или `int`)? Существует два разных подхода: «big-endian» и «little-endian».* Первый предполагает, что сначала должны следовать старшие байты, а второй — наоборот. Это важно, если двоичные данные, которые вы записываете, предназначены

* Термины «big-endian» («тупоконечник») и «little-endian» («остроконечник») заимствованы из произведения Джонатана Свифта «Путешествия Гулливера», в котором лилипуты были разделены на два враждующих лагеря, споривших о том, с какого конца нужно разбивать вареные яйца — тупого (big end) или острого (little end).

для потребления скомпилированной программой. В Intel-совместимых компьютерах используется вариант little-endian, в то время как множество рабочих станций под управлением операционной системы Unix применяют big-endian. Выбор между этими двумя концепциями инкапсулируется с помощью класса `ByteOrder`. Чтобы задать порядок следования байтов для объекта `ByteBuffer`, можно воспользоваться методом `order()` в сочетании с идентификаторами `ByteOrder.BIG_ENDIAN` и `ByteOrder.LITTLE_ENDIAN`. Например:

```
byteArray.order( ByteOrder.BIG_ENDIAN );
```

С помощью метода `ByteOrder.nativeOrder()` вы можете узнать стандартный порядок следования байтов для вашей платформы (я не сомневаюсь в вашей любознательности).

Выделение места для буфера

Место для буфера можно выделять явно или отталкиваясь от уже имеющегося массива. Любой буфер содержит два статических метода; первый, `allocate()`, принимает значение вместимости (размер), а второй, `wrap()`, в качестве аргумента имеет существующий массив:

```
CharBuffer cbuf = CharBuffer.allocate( 64*1024 );
```

Схожим образом в памяти размещается и прямой буфер, только для него используется метод `allocateDirect()`:

```
ByteBuffer bbuf = ByteBuffer.allocateDirect( 64*1024 );
ByteBuffer bbuf2 = ByteBuffer.wrap( someExistingArray );
```

Как уже было сказано ранее, благодаря использованию структур памяти из операционной системы прямые буферы способны оптимизировать некоторые операции ввода/вывода. С одной стороны, это повышает производительность, но с другой — немного замедляет процесс выделения места и «утяжеляет» методы для работы с данными. В связи с этим прямые буферы лучше подходят для продолжительных задач.

Кодирование и декодирование символов

Для превращения символов в обычные байты (и наоборот) существуют специальные компоненты, кодировщики и декодеры, которые используют стандарт Unicode как промежуточную кодировку. Они давно входят в состав платформы Java и применяются в потоках типа

Reader и Writer, а также в методах класса String, которые имеют дело с массивами. Тем не менее на ранних этапах развития платформы Java с кодировками нельзя было работать напрямую; при необходимости на кодировщики и декодеры ссылались по их именам (в виде строк). Идея набора кодировок стандарта Unicode нашла свое воплощение в пакете java.nio.charset в виде класса Charset.

Класс Charset является фабрикой объектов одноименного типа, каждый из которых знает, как кодировать символьные буферы в байтовые и как декодировать их обратно. Чтобы найти кодировку, можно передать ее имя в статический метод Charset.forName(), и затем использовать полученный результат для выполнения преобразований:

```
Charset charset = Charset.forName("US-ASCII");
CharBuffer charBuff = charset.decode( byteBuff ); // в ascii
ByteBuffer byteBuff = charset.encode( charBuff ); // и обратно
```

Доступность любой кодировки можно проверить с помощью метода Charset.isSupported().

В стандартную поставку входит следующий набор кодировок:

- US-ASCII
- ISO-8859-1
- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-16

Вы можете получить список всех кодировок, доступных на вашей платформе, используя метод availableCharsets():

```
Map map = Charset.availableCharsets();
Iterator it = map.keySet().iterator();
while ( it.hasNext() )
   System.out.println( it.next() );
```

Результатом будет ассоциативный массив, так как кодировки могут иметь «псевдонимы» и упоминаться под разными названиями.

Помимо классов из пакета java.nio, ориентированных на работу с буферами, поддержка кодировок была добавлена в промежу-

точные классы классического API-интерфейса, `InputStreamReader` и `OutputStreamWriter`. Указать кодировку можно по ее названию или в виде объекта `Charset`.

Классы `CharsetEncoder` и `CharsetDecoder`

Объекты `CharsetEncoder` и `CharsetDecoder` помогут вам лучше контролировать процесс кодирования и декодирования (вместе они представляют собой кодек); для их создания можно воспользоваться методами `newEncoder()` и `newDecoder()` из объекта `Charset`. В предыдущем примере мы исходили из того, что вся информация находится в едином буфере. Однако чаще всего данные приходится обрабатывать по мере их поступления, отдельными порциями. API-интерфейс для кодирования/декодирования позволяет делать это при помощи методов `encode()` и `decode()`; эти методы принимают модификатор, который указывает на то, нужно ли ожидать появления новых данных (без этой информации кодек может зависнуть в процессе преобразования многобайтовых символов, если данные закончатся). Зная о том, что данные продолжают поступать, кодек не станет генерировать ошибку, пока не завершится преобразование. В следующем фрагменте кода мы используем декодер для чтения байтов из буфера `bbuff` (`ByteBuffer`) и передачи полученных символов в буфер `cbuff` (`CharBuffer`):

```
CharsetDecoder decoder = Charset.forName("US-ASCII").newDecoder();
boolean done = false;
while ( !done ) {
   bbuff.clear();
   done = ( in.read( bbuff ) == -1 );
   bbuff.flip();
   decoder.decode( bbuff, cbuff, done );
}
cbuff.flip();
// используем cbuff. . .
```

Чтобы установить модификатор `done`, мы проверяем, выполняется ли входное условие для канала `in`. Обратите внимание, что с помощью метода `flip()` из объекта `ByteBuffer` мы устанавливаем максимальный объем данных, предназначенных для чтения, и сбрасываем текущую позицию, что дает нам возможность выполнить декодирование в рамках одного вызова. Методы `encode()` и `decode()` возвращают объект `CoderResult`, который позволяет следить за ходом кодирова-

ния (в нашем примере мы его не используем). С помощью методов `isError()`, `isUnderflow()` и `isOverflow()` из объекта `CoderResult` можно узнать, что именно привело к остановке процесса кодирования: ошибка, нехватка байтов во входящем буфере или переполнение исходящего буфера.

Класс `FileChannel`

Разобравшись с основными аспектами каналов и буферов, мы можем приступить к рассмотрению реального компонента. Класс `FileChannel` из пакета NIO является эквивалентом класса `java.io.RandomAccess`, но помимо улучшенной производительности он обладает несколькими новыми важными возможностями.

Вы можете использовать его вместо файловых потоков из пакета `java.io`, если вам необходимы такие функции, как блокирование файлов, доступ к файлам, хранящимся в оперативной памяти, или высоко оптимизированная передача данных (возможно, между файловыми или сетевыми каналами).

Объект `FileChannel` можно создать на основе пути `Path`, используя статический метод `FileChannelopen()`.

```
FileSystem fs = FileSystems.getDefault();
Path p = fs.getPath( "/tmp/foo.txt" );
// Открываем канал для чтения в стандартном режиме
try ( FileChannel channel = FileChannel.open(( p ) ) {
   ...
}
// Открываем канал для записи, используя опции
import static java.nio.file.StandardOpenOption.*;
try ( FileChannel channel = FileChannel.open( p, WRITE, APPEND, ... ) ) {
   ...
}
```

По умолчанию метод `open()` создает для файла канал, который поддерживает только чтение. Но мы можем открыть канал для записи или добавления данных, а также управлять более тонкими функциями, такими как атомарное создание или синхронизация. Как видно из второй части вышеприведенного примера, это делается с помощью дополнительных опций, полный список которых приведен в табл. 12.4.

Табл. 12.4. Опции перечисления `java.nio.file.StandardOpenOption`

Опция	Описание
`READ`, `WRITE`	Файл открывается только для чтения (по умолчанию) или только для записи. Для чтения/записи используйте обе опции.
`APPEND`	Файл открывается для записи; все данные записываются в конец файла.
`CREATE`	Используется в сочетании с опцией `WRITE`, чтобы файл не только открывался, но и создавался, если это нужно.
`CREATE_NEW`	Используется в сочетании с опцией `WRITE`, чтобы файл создавался автоматически; операция заканчивается неудачно, если файл уже существует.
`DELETE_ON_CLOSE`	Происходит попытка удаления файла, если он закрыт; если файл остается открытым, удаление выполняется при завершении работы Java-машины.
`SYNC`, `DSYNC`	Гарантируют блокирование файла (если это возможно), пока все данные не будут записаны на диск. Опция `SYNC` делает это для любых изменений, включая данные и метаданные (атрибуты); `DSYNC` отвечает только за содержимое файла.
`SPARSE`	Используется, если создаваемый файл должен быть разреженным. Разреженные файлы обычно имеют большой размер и содержат информацию о пустых фрагментах, которые не занимают реальное место на диске (поддерживается не во всех файловых системах).
`TRUNCATE_EXISTING`	Эквивалентно использованию опции `WRITE` для существующего файла; при открытии длина файла обнуляется.

Канал `FileChannel` также можно получить из таких классических классов как `FileOutputStream` или `RandomAccessFile`:

```
FileChannel readOnlyFc = new FileInputStream("file.txt").getChannel();
FileChannel readWriteFc = new RandomAccessFile("file.txt", "rw")
   .getChannel();
```

Объекты `FileChannel`, созданные с помощью файлового ввода или исходящего потока, доступны, соответственно, только для чтения или только записи. Чтобы получить канал для чтения *и* записи, при создании объекта `RandomAccessFile` необходимо указать соответствующие права доступа, как было сделано в предыдущем примере.

Объект `FileChannel` используется так же, как и `RandomAccessFile`, только вместо массива байтов он работает с буфером `ByteBuffer`:

```
ByteBuffer bbuf = ByteBuffer.allocate( ... );
bbuf.clear();
readOnlyFc.position( index );
```

```
readOnlyFc.read( bbuf );
bbuf.flip();
readWriteFc.write( bbuf );
```

Контролировать количество прочитанных или записанных данных можно путем регулирования текущей позиции буфера в сочетании с границей/меткой; можно также воспользоваться альтернативными версиями методов для чтения/записи, которые принимают начальную позицию буфера и его длину. Чтобы выполнить чтение или запись в произвольной позиции, нужно передать индекс:

```
readWriteFc.read( bbuf, index )
readWriteFc.write( bbuf, index2 );
```

Какой бы вариант вы ни выбрали, количество прочитанных или записанных байтов зависит от нескольких факторов. Операции чтения/записи пытаются дойти до границы буфера, и в случае с локальными файлами им это почти всегда удается. Они гарантированно блокируются до тех пор, пока не будет обработан хотя бы один байт. В любом случае в качестве результата возвращается количество обработанных байтов; при этом соответствующим образом обновляется текущая позиция буфера, что позволяет повторять операцию до тех пор, пока данные не закончатся. Это одно из преимуществ работы с буферами — они сами проводят подсчет байтов. По аналогии со стандартными потоками, когда ввод заканчивается, метод `read()`, принадлежащий каналу, возвращает –1.

Объем доступных данных всегда можно узнать с помощью метода `size()`. Этот показатель может меняться, если вы записываете данные в конец файла. Кроме того, вы можете урезать файл до определенной длины, используя метод `truncate()`.

Параллельный доступ

Канал `FileChannel` можно безопасно использовать сразу в нескольких потоках выполнения; он гарантирует, что данные, которые «видит» каждый поток, согласуются между собой в пределах одной Java-машины. Но при этом невозможно предвидеть, с какой скоростью будет выполняться запись данных в хранилище (если не использовать опции `SYNC` или `DSYNC`). Если целостность данных нужно обеспечивать только время от времени, вы можете использовать метод `force()`, чтобы сбрасывать изменения на диск. Аргумент типа `Boolean`, который принимает

этот метод, определяет, нужно ли синхронизировать метаданные файла, включая время изменения и права доступа (то же самое, что с опциями `SYNC` и `DSYNC`). Некоторые системы отслеживают любой доступ к файлу, будь то запись или чтение, поэтому, если вам не нужна мгновенная синхронизация данных, лучше передать методу `force()` значение `false` — это значительно сократит количество обновлений файла.

Как и любой другой канал, объект `FileChannel` может быть закрыт из любого потока выполнения. После закрытия вызов любого метода, связанного с чтением/записью и текущей позицией, будет генерировать исключение `ClosedChannelException`.

Блокирование файлов

Канал `FileChannel` поддерживает эксклюзивное и разделяемое блокирование отдельных частей файла посредством метода `lock()`:

```
FileLock fileLock = fileChannel.lock();
int start = 0, len = fileChannel2.size();
FileLock readLock = fileChannel2.lock( start, len, true );
```

Блокировки могут быть либо разделяемыми, либо эксклюзивными. *Эксклюзивная* блокировка условно гарантирует, что никто больше не сможет заблокировать файл или часть файла. *Разделяемая* блокировка позволяет нескольким компонентам выполнять одновременное блокирование (но только в разделяемом режиме). Первый вид блокировок подходит для записи, а второй — для чтения. Когда вы будете записывать данные, вам точно не захочется, чтобы кто-то другой мог выполнять запись одновременно с вами; параллельное чтение вполне допустимо, но, опять же, при этом нужно исключить возможность записи.

Версия метода `lock()`, которая не принимает аргументов (см. предыдущий пример), пытается выполнить эксклюзивную блокировку для всего файла. Вторая версия принимает начальную позицию, длину и параметр, который определяет, должна ли блокировка быть разделяемой (или эксклюзивной). С помощью объекта `FileLock`, который возвращает метод `lock()`, можно снять блокировку:

```
fileLock.release();
```

Стоит заметить, что блокирование файлов работает только в том случае, если API-интерфейс используют *все* стороны; блокировка не запрещает кому-либо считывать или записывать содержимое файла.

По большому счету, единственный способ гарантировать выполнение условий блокирования — всегда пытаться создавать блокировку перед доступом к файлу. Кроме того, разделяемое блокирование не поддерживается некоторыми системами; в таких случаях любая блокировка становится эксклюзивной. Вид блокировки можно проверить с помощью метода `isShared()`.

Блокирование действует до тех пор, пока канал `FileChannel` не будет закрыт или разорван, поэтому, чтобы процесс освобождения ресурсов был более надежным, блокировку лучше создавать внутри конструкции try-with-resources.

```
try ( FileChannel channel = FileChannel.open( p, WRITE ) ) {
   channel.lock();
   ...
}
```

Отображение файлов в память

Возможность отображения файла в память — это одна из наиболее интересных функций класса `FileChannel`. Она позволяет работать с файлом как с единым буфером `ByteBuffer` — словно все его содержимое магическим образом было целиком перенесено в оперативную память. Эта функция имеет чрезвычайно эффективную реализацию, являясь одним из самых быстрых средств доступа к файлам. Отображение файлов в память может сохранить множество времени и ресурсов при работе с большими объемами данных.

Это может показаться подозрительным, ведь мы получаем принципиально более простой способ доступа к данным, который, к тому же, работает быстрее и эффективнее. В чем подвох? На самом деле никакого подвоха нет. Причина в том, что в основе любой современной операционной системы лежит концепция виртуальной памяти. Если коротко, это означает, что некоторая часть дискового пространства может вести себя, как оперативная память; это достигается за счет постоянной подкачки страниц («страницей» зовется блок данных размеров 4 Кб) с диска в память и наоборот, которая выполняется абсолютно прозрачно для приложений. Операционные системы умеют делать это очень хорошо; они эффективно кешируют данные, которые используются программой, освобождая те ресурсы, которые больше не нужны. Отображая файлы в память, Java-машина просто использует внутренние механизмы ОС.

Хорошим примером того, где может пригодиться такой подход, является база данных. Представьте, что файл размером 10 Гб содержит записи, проиндексированные в разных местах. Если файл будет находиться в памяти, мы сможем использовать стандартный объект `ByteBuffer`, чтобы считывать и записывать произвольные участки буфера; при этом данные по мере необходимости будут разбиваться на страницы, а для их чтения/записи будут использоваться средства операционной системы. Мы могли бы эмулировать этот процесс при помощи классов `RandomAccessFile` или `FileChannel`, но тогда бы нам пришлось вручную перемещать данные в буфер, и такая реализация практически наверняка была бы менее эффективной.

Процесс отображения выполняется с помощью метода `map()` из объекта `FileChannel`. Например:

```
FileChannel fc =FileChannel.open( fs.getPath("index.db"), CREATE, READ,
WRITE );
MappedByteBuffer mappedBuff =
   fc.map( FileChannel.MapMode.READ_WRITE, 0, fc.size() );
```

Метод `map()` возвращает объект `MappedByteBuffer`, который почти ничем не отличается от стандартного буфера `ByteBuffer`, если не считать нескольких дополнительных методов, связанных с отображением файлов. Самым важным из них является метод `force()`; все данные, которые были записаны в буфер, он сбрасывает в постоянное хранилище на диске. Константы-идентификаторы `READ_ONLY` и `READ_WRITE` из внутреннего статического класса `Channel.MapMode` определяют тип доступа. Доступ на чтение/запись возможен только в том случае, если аналогичным доступом обладает канал, задействованный для отображения. Данные, которые считываются из буфера, всегда целостны в рамках одной Java-машины. Их целостность может соблюдаться и для разных приложений на одном компьютере, но этого никто не гарантирует.

Напомним, что буферы `MappedByteBuffer` и `ByteBuffer` работают одинаково. В продолжение предыдущего примера мы можем декодировать содержимое буфера с помощью символьного декодера и поискать текст, который соответствует регулярному выражению:

```
CharBuffer cbuff = Charset.forName("US-ASCII").decode( mappedBuff );
Matcher matcher = Pattern.compile("abc*").matcher( cbuff );
while ( matcher.find() )
   System.out.println( matcher.start()+": "+matcher.group(0) );
```

Воспользовавшись API-интерфейсом для регулярных выражений, мы реализовали некое подобие команды *grep* из операционной системы Unix; при этом мы работали с буфером `CharBuffer` так, словно это последовательность `CharSequence`. В этом примере мы немного схитрили, так как буфер `CharBuffer`, который возвращается методом `decode()`, имеет такой же размер, как и отображенный файл, а значит, должен помещаться в память. Для более эффективного выполнения данной задачи мы могли бы воспользоваться объектом `CharsetDecoder`, который был рассмотрен ранее в этой главе; это позволило бы нам обрабатывать большие файлы, не загружая их в память целиком.

Передача данных напрямую

Напоследок мы обсудим оптимизацию производительности, которую предлагает канал `FileChannel`. Речь пойдет о двух хорошо оптимизированных методах, `transferFrom()` и `transferTo()`, которые позволяют перемещать данные между файловым и каким-либо другим каналом. Эти методы пользуются преимуществами прямых буферов, что часто позволяет им передавать информацию максимально быстро, минуя этап копирования байтов в память Java-машины. Следующий пример, вероятно, является самой быстрой реализацией копирования файлов на языке Java, если не считать встроенного метода `Files.copy()`:

```
import java.nio.channels.*;
import java.nio.file.*;
import static java.nio.file.StandardOpenOption.*;
public class CopyFile
{
   public static void main( String [] args ) throws Exception
   {
      FileSystem fs = FileSystems.getDefault();
      Path fromFile = fs.getPath( args[0] );
      Path toFile = fs.getPath( args[1] );
      try (
         FileChannel in = FileChannel.open( fromFile );
         FileChannel out = FileChannel.open( toFile, CREATE, WRITE ); )
      {
         in.transferTo( 0, (int)in.size(), out );
      }
   }
}
```

Класс AsynchronousFileChannel

Когда мы опять вернемся к пакету NIO в следующей главе, вы увидите, что сетевые каналы являются разновидностями типа `SelectableChannel`; это означает, что ими можно управлять с помощью *селекторов*. Селекторы позволяют проверять, готов ли канал для чтения или записи, благодаря чему можно не блокировать потоки выполнения и работать эффективней. Файловые каналы *не* являются выборочными, поэтому большинство операций с файлами оставляют поток блокированным, пока не закончат работу. Это не означает, что мы должны ждать записи или считывания всех байтов. Обычно операции чтения/записи возвращают меньше байтов, чем у них запрашивали; кроме того, если при открытии файла не были указаны опции `SYNC` или `DSYNC`, запись может выполняться с помощью буфера, размещенного в памяти. Но в нашем мире, где жесткие диски на много-много порядков уступают в скорости оперативной памяти, даже частичные чтение или запись могут оказаться достаточно медленными, чтобы у нас возникло желание избавиться от блокирующих операций.

В качестве очевидного решения напрашивается использование многопоточности и управление операциями чтения и записи в специальном потоке выполнения, отдельно от основной бизнес-логики. Однако эту задачу можно решить еще проще благодаря классу `AysnchronousFileChannel`, который появился в платформе Java 7. Это файловый канал, который переадресовывает все свои операции в пул потоков, а результат возвращает в виде объекта `Future` или с помощью асинхронных функций обратного вызова. Для всех методов чтения/записи этого канала необходимо указывать сдвиг в байтах (так как в случае асинхронной работы понятие «текущего» сдвига теряет свой смысл). В качестве простейшего примера мы обновим файл в фоновом режиме, без получения результатов:

```
AsynchronousFileChannel channel = AsynchronousFileChannel.open( path,
WRITE );
// Записываем буфер logBuffer в конец файла в фоновом
// режиме, не дожидаясь результата.
channel.write( logBuffer, channel.size() );
...
```

Здесь вместо обычного файлового канала мы создали объект `AsynchronousFileChannel`. Запись выполняется в фоновом режиме, а метод `write()` сразу же завершается. По умолчанию для выполне-

ния фоновой записи наш канал будет использовать системный пул потоков. Но при вызове метода `open()` мы могли бы указать наш собственный пул, воспользовавшись сервисом `Executor`. Если в какой-то момент нам понадобится синхронизироваться, чтобы все данные гарантированно записались, мы сможем вызвать метод `force()`; это позволит заблокировать поток выполнения и подождать окончания записи.

Операция чтения выглядит более интересной, ведь она должна возвращать прочитанные байты. В этом случае мы можем предоставить функцию обратного вызова в виде объекта `CompletionHandler`, который вернет нам результаты, когда они будут готовы.

```
AsynchronousFileChannel channel = AsynchronousFileChannel.open( path );
ByteBuffer bbuff = ByteBuffer.allocate( 1024 );
Object attachment = ...;
channel.read( bbuff, offset, attachment,
   new CompletionHandler<Integer, Object>() {
      @Override
      public void completed( Integer result, Object attachment ) {
      System.out.println( "прочитанных байтов: " + result );
      }
      @Override
      public void failed( Throwable exc, Object attachment ){
      ...
      }
   } );
```

В качестве дополнительного аргумента `attachment` для метода `read()` может выступать любой объект на наш выбор; это контекст, который возвращается внутри функции обратного вызова и необходим для обслуживания результата. Здесь мы выводим количество прочитанных байтов, которое может оказаться меньшим, чем мы запрашивали (но, по крайней мере, нам не пришлось ждать, пока они появятся). Если операция чтения завершится неудачно, сработает метод `failed()` с соответствующим исключением.

Масштабируемый ввод/вывод и пакет NIO

В этой главе мы заложили фундамент для дальнейшего применения пакета NIO, но некоторые важные моменты остались нерассмотренными.

В следующей главе, когда речь пойдет о неблокирующем и выборочном вводе/выводе, мы увидим еще больше реальных примеров использования данной технологии. Помимо улучшения производительности, которого можно добиться за счет прямых буферов, пакет `java.nio` предлагает возможность построения серверных приложений, которые способны эффективно масштабироваться для уровня больших систем, используя при этом меньшее количество потоков. Не останутся без внимания и другие важные типы каналов: `SocketChannel`, `ServerSocketChannel` и `DatagramChannel`.

Глава 13
СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

Сеть — это сердце платформы Java. Почти все ее самые интересные особенности связаны с возможностью разработки динамических приложений с сетевой структурой. По мере того, как сетевые API-интерфейсы стандартной библиотеки становились более зрелыми, язык Java все чаще использовался для реализации традиционных клиент-серверных программ и сервисов. Мы начнем эту главу с обсуждения пакета `java.net`, который содержит фундаментальные классы для передачи информации и работы с сетевыми ресурсами (эта тема будет актуальна и в главе 14). Затем мы сосредоточимся на пакете `java.rmi`, который предоставляет стандартный высокоуровневый интерфейс для удаленного вызова методов (Remote Method Invocation или RMI). В конце, отталкиваясь от материала, изученного в главе 12, мы завершим рассмотрение пакета `java.nio` — высокоэффективного средства для реализации крупных серверов.

Классы пакета `java.net` можно разделить на две главных категории: API-интерфейс для работы с низкоуровневыми интернет-протоколами (сокеты) и высокоуровневые API-интерфейсы для создания веб-приложений, ориентированные на работу с едиными указателями ресурсов (Uniform Resource Locator или URL). Структура пакета `java.net` представлена на рис. 13.1.

API-интерфейс для работы с сокетами предоставляет доступ к стандартным сетевым протоколам, которые используются для связи между разными компьютерами в Интернете. Сокеты лежат в основе любого сетевого взаимодействия. Это самый низкий уровень в иерархии основных сетевых абстракций — с помощью сокетов вы можете устанавливать связь между клиентскими и серверными (или равноценными) приложениями в сети, но для этого вам придется реализовать свой собственный программный протокол, который будет обрабатывать и интерпретировать данные. Поверх сокетов работают любые сетевые инструменты высокого уровня, включая средства уделанного вызова методов, протокол HTTP и веб-сервисы.

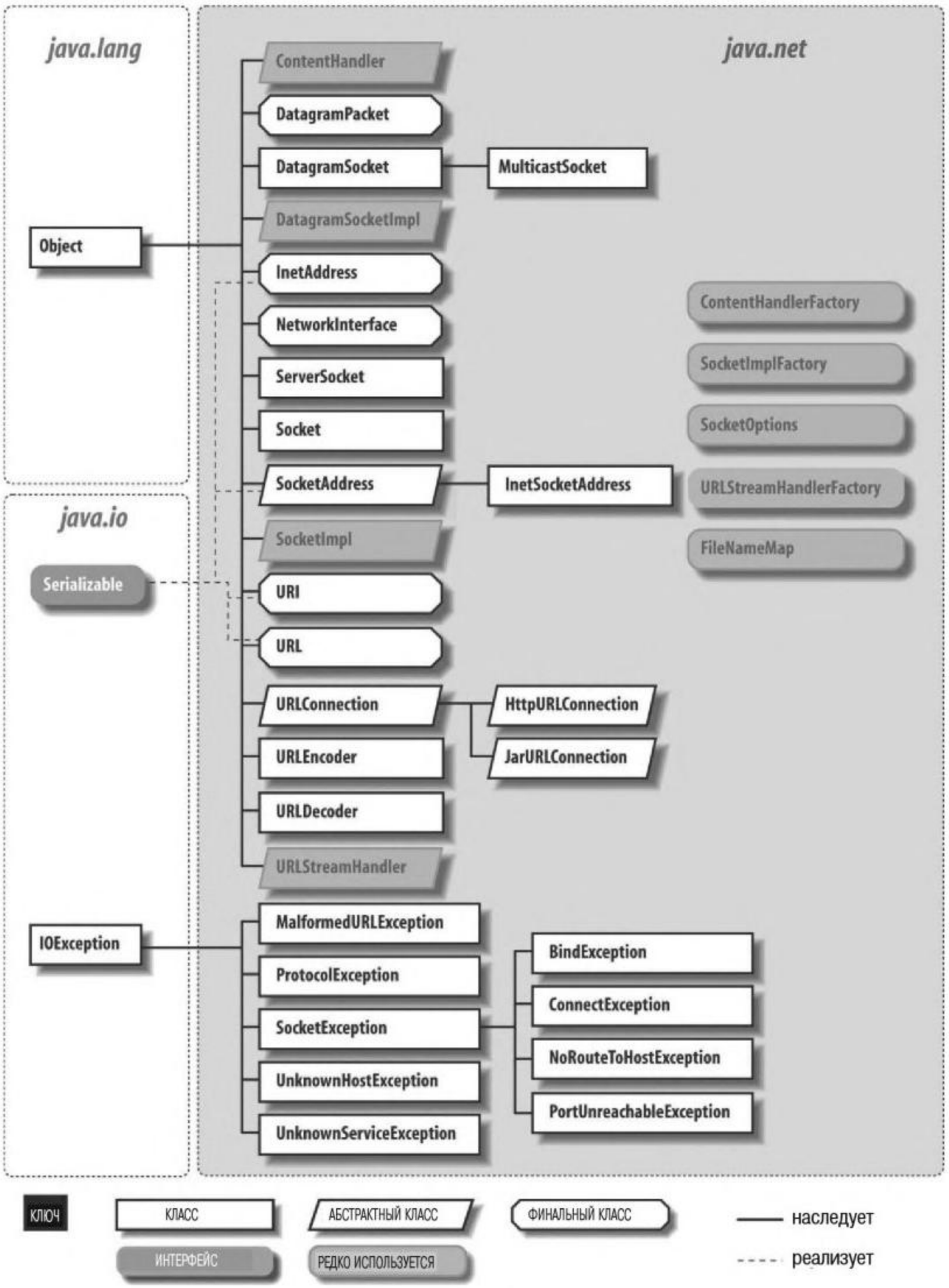

Рис. 13.1. Пакет `java.net`

Технология RMI представляет собой мощный инструмент, который пользуется всеми преимуществами сериализации. Она позволяет прозрачно работать с объектами языка Java, которые находятся на удаленных компьютерах, так, как будто они доступны локально. Технология RMI

упрощает написание распределенных приложений, в которых клиенты и серверы работают с общими данными как с полноценными объектами языка, а не как с обычными потоками или пакетами информации.

В отличие от технологии RMI, которая может использоваться только между двумя Java-приложениями, *веб-сервисы* являются более универсальным средством, не зависящим от платформы. Они опираются на такие веб-стандарты как HTML и XML, позволяя вызывать слабосвязанные функции, предоставляемые удаленными серверами. Мы вернемся к веб-сервисам в главах 14 и 15, когда приступим к изучению веб-программирования.

В этой главе мы рассмотрим несколько простых и практичных примеров высокоуровневого и низкоуровневого программирования на языке Java с использованием сокетов и технологии RMI. Глава 14 посвящена другой части пакета `java.net`, которая позволяет клиентским приложениям работать с веб-серверами и сервисами посредством URL-адресов. Глава 15 охватывает сервлеты и инструменты, которые позволяют создавать веб-приложения и сервисы для веб-серверов.

Сокеты

Сокеты — это низкоуровневый программный интерфейс для сетевого взаимодействия. Они передают потоки данных между приложениями, которые могут находиться как на одном компьютере, так и в разных частях сети. Сокеты впервые появились в семействе операционных систем BSD Unix; в некоторых языках программирования они чрезвычайно усложнены и состоят из множества мелких компонентов, которые в любой момент могут рассыпаться на части и погубить всю программу. Дело в том, что большинство API-интерфейсов для работы с сокетами можно использовать в сочетании с практически любыми сетевыми протоколами. А так как процесс передачи данных по сети может выполняться совершенно разными способами, интерфейсы сокетов часто бывают довольно сложными*.

Пакет `java.net` поддерживает упрощенные, объектно-ориентированные интерфейсы сокетов, которые значительно облегчают сетевое взаимодействие. Если у вас уже имеется опыт написания сетевых при-

* Более общую информацию о сокетах можно найти в книге Ричарда Стивенса «Unix. Взаимодействие процессов» («Питер», 2002).

ложений на других языках, вы будете приятно удивлены тем, насколько простым может оказаться этот процесс, когда все зубодробительные подробности скрыты внутри объектов. Если это ваше первое знакомство с сокетами, вы убедитесь в том, что общение с другими приложениями по сети может выглядеть не сложнее, чем чтение файла или получение пользовательского ввода. Большинство средств ввода/вывода на платформе Java (включая сетевые) используют классы потоков, описанные в главе 12. Потоки предоставляют унифицированный интерфейс ввода/вывода, благодаря которому операции чтения и записи работают одинаково как в Интернете, так и на локальной файловой системе. Помимо потоковых интерфейсов сетевая инфраструктура платформы Java предоставляет компоненты на основе буферов из пакета NIO с прекрасной поддержкой масштабирования. В этой главе мы рассмотрим обе концепции.

В стандартной библиотеке содержится три разновидности сокетов для поддержки трех разных низкоуровневых протоколов: `Socket`, `DatagramSocket` и `MulticastSocket`. В этом разделе мы сосредоточимся на базовом классе `Socket`, который использует *надежный* протокол с *установлением логических соединений*. Он работает по принципу телефонных звонков. Установив соединение, две программы могут обмениваться потоками информации; при этом само соединение является постоянным и удерживается даже тогда, когда обмена данными не происходит. Это надежный протокол — он следит за тем, чтобы данные не терялись (отправляя их повторно, если потребуется) и чтобы байты информации были получены именно в том порядке, в котором вы их отправляли.

В следующем разделе мы рассмотрим класс `DatagramSocket`, который использует *ненадежный* протокол *без установления соединений*. Здесь в качестве аналогии напрашивается обычная почта. Программы могут обмениваться друг с другом короткими сообщениями, не устанавливая при этом прямых соединений и не соблюдая порядок отправки информации. К тому же нет никакой гарантии, что сообщения вообще дойдут до адресата. Являясь разновидностью сокета `DatagramSocket`, класс `MulticastSocket` занимается групповой передачей — одновременной отправкой данных множеству получателей. С точки зрения использования эти два вида сокетов мало чем отличаются друг от друга. Но в виду того, что многоадресная передача в современном Интернете применяется не так широко, класс `MulticastSocket` здесь рассматриваться не будет.

Теоретически в основе сокетов может лежать почти любой протокол (бывалые компьютерщики должны помнить такие названия как Novell IPX, Apple AppleTalk и т. д.). Но на практике есть всего одно существен-

ное семейство протоколов, которое используется в Интернете и поддерживается платформой Java; центральное место в этом семействе занимает *межсетевой протокол* (Internet Protocol или IP). Класс `Socket` общается по протоколу TCP (Transmission Control Protocol — протокол управления передачей), который является разновидностью IP с поддержкой соединений; класс `DatagramSocket`, не использующий соединения, основан на протоколе UDP (User Datagram Protocol — протокол пользовательских датаграмм).

Клиенты и серверы

При разработке сетевых приложений обычно используют такие понятия, как «клиент» и «сервер». Граница между ними довольно размытая, хотя обычно та сторона, которая инициирует взаимодействие, считается *клиентом*. Сторону, принимающую запрос, чаще всего называют *сервером*. В случае если сокеты используются в равноценных приложениях, это разделение не так важно, но чтобы не запутаться, будем придерживаться вышеозначенного определения.

С практической точки зрения главная разница между клиентом и сервером заключается в том, что клиент в любой момент времени может создать сокет, который инициирует связь с серверным приложением, тогда как сервер должен быть заранее готовым принимать входящие подключения. Класс `java.net.Socket` представляет одну сторону каждого отдельного соединения как на клиенте, так и на сервере. Кроме того, сервер использует класс `java.net.ServerSocket`, чтобы отслеживать попытки подключения со стороны клиентов. В большинстве случаев приложение, которое считается сервером, создает объект `ServerSocket` и ждет, когда кто-то подключится (метод `accept()` блокирует поток выполнения). Дождавшись соединения, метод `accept()` возвращает объект `Socket`, с помощью которого сервер общается с клиентом. Сервер может взаимодействовать одновременно с множеством клиентов, каждому из которых потребуется отдельная копия `Socket`, хотя объект `ServerSocket` при этом может оставаться в единственном экземпляре (см. рис. 13.2).

С точки зрения сокетов клиенту необходимо знать две вещи, чтобы найти сервер в Интернете и подключиться к нему: *имя сервера* (используется для определения сетевого адреса компьютера) и *номер порта*. Номер порта — это идентификатор, который помогает различать разные клиентские и серверные приложения, запущенные на одном компьютере. Когда сервер ждет подключения, он «прослушивает» заранее

выбранный порт. Номер порта, который назначен сервису, используется клиентами для соединения. Представьте, что компьютер — это отель, а приложения, которые на нем работают — постояльцы; тогда порт — это табличка с цифрами на дверях каждого отдельного номера. Чтобы связаться с постояльцем отеля, вам нужно знать его имя и номер, в котором он проживает.

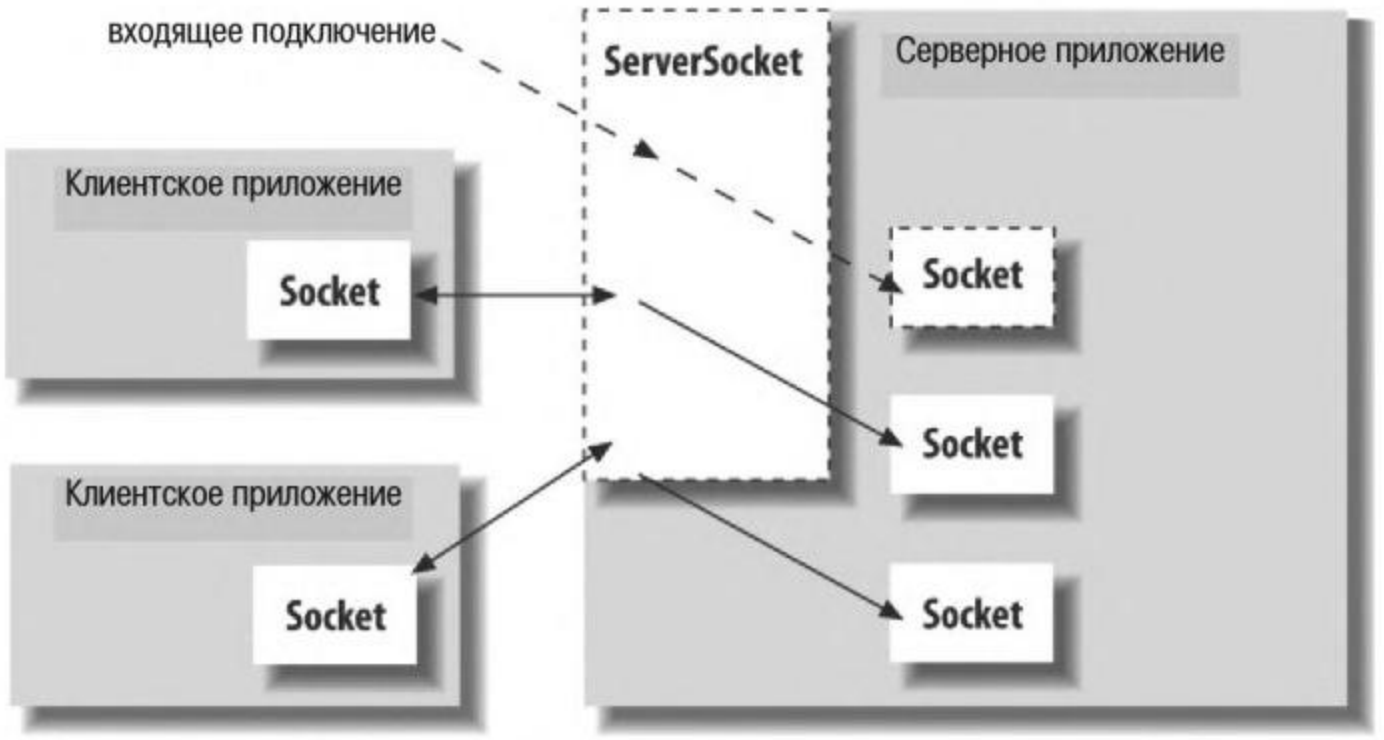

Рис. 13.2. Клиенты и серверы в виде объектов `Socket` и `ServerSocket`

Клиенты

Чтобы открыть соединение с сервером, клиентское приложение создает объект `Socket`, указывая имя компьютера и подходящий порт:

```
try {
   Socket sock = new Socket("smtp.yandex.ru", 25);
} catch ( UnknownHostException e ) {
   System.out.println("Не удается найти сервер.");
} catch ( IOException e ) {
   System.out.println("Ошибка при подключении к серверу.");
}
```

Данный фрагмент кода пытается подключить сокет к порту под номером 25 (почтовый сервис с протоколом SMTP), который принадлежит серверу **smtp.yandex.ru**. Клиент учитывает вероятность того, что имя сервера может быть не найдено (исключение `UnknownHostException`), и что к нему не удастся подключиться (исключение `IOException`). В первом случае платформа Java пытается получить IP-адрес для указанного имени сервера, используя стандартный сервис DNS (Domain Name

System — система доменных имен). Конструктору сразу можно передать строку с подходящим IP-адресом:

```
Socket sock = new Socket("213.180.193.38", 25);
```

После установления соединения можно получить входящий и исходящий потоки, воспользовавшись методами `getInputStream()` и `getOutputStream()` из объекта `Socket`. Следующий код (который выбран исключительно в качестве примера) использует эти потоки для отправки и получения данных:

```
try {
   Socket server = new Socket("foo.bar.com", 1234);
   InputStream in = server.getInputStream();
   OutputStream out = server.getOutputStream();
   // записываем байт
   out.write(42);
   // записываем строку, заканчивающуюся символом
   // перевода строки или возврата каретки
   PrintWriter pout = new PrintWriter( out, true );
   pout.println("Привет!");
   // считываем байт
   byte back = (byte)in.read();
   // считываем строку, заканчивающуюся символом
   // перевода строки или возврата каретки
   BufferedReader bin =
      new BufferedReader( new InputStreamReader( in ) );
   String response = bin.readLine();
   // отправляем сериализованный объект
   ObjectOutputStream oout = new
   ObjectOutputStream( out );
   oout.writeObject( new java.util.Date() );
   oout.flush();
   server.close();
}
catch (IOException e ) { ... }
```

Этот пример начинается с того, что клиент создает объект `Socket` для связи с сервером. В конструкторе класса `Socket` указывается имя сервера **(foo.bar.com)** и заранее выбранный номер порта **(1234)**. Установив соединение, клиент передает серверу один байт, используя метод `write()` из потока `OutputStream`. (Чтобы отправить вместо байта текстовую строку,

лучше заключить поток `OutputStream` внутрь объекта `PrintWriter`.) Далее клиент выполняет обратную операцию: получает байт от сервера, используя метод `read()` из потока `InputStream`, и затем создает буфер `BufferedReader`, из которого извлекает полноценную строку. В конце мы делаем кое-что по-настоящему виртуозное: отправляем на сервер сериализованный объект, используя поток `ObjectOutputStream`. (Позже в этой главе мы рассмотрим все подробности передачи сериализованных объектов.) После этого клиент разрывает соединение с помощью метода `close()`. Любая из этих операций может сгенерировать исключение `IOException`; на этот случай в нашем приложении предусмотрен блок `catch`.

Серверы

После налаживания соединения серверное приложение начинает использовать на своей стороне обычный экземпляр `Socket`. Но чтобы принять подключение от клиента, нужно сначала создать объект `ServerSocket`, привязанный к нужному порту. Давайте воссоздадим сетевое общение из предыдущего примера, но уже с точки зрения сервера:

```
// А тем временем на сервере foo.bar.com...
try {
   ServerSocket listener = new ServerSocket( 1234 );
   while ( !finished ) {
      Socket client = listener.accept(); // ждем подключения
      InputStream in = client.getInputStream();
      OutputStream out = client.getOutputStream();
      // считываем байт
      byte someByte = (byte)in.read();
      // считываем строку, заканчивающуюся символом
      // перевода строки или возврата каретки
      BufferedReader bin =
         new BufferedReader( new InputStreamReader( in ) );
      String someString = bin.readLine();
      // записываем байт
      out.write(43);
      // прощаемся
      PrintWriter pout = new PrintWriter( out, true );
      pout.println("Всего хорошего!");
      // считываем сериализованный объект
      ObjectInputStream oin = new ObjectInputStream( in );
```

```
        Date date = (Date)oin.readObject();
        client.close();
    }
    listener.close();
}
catch (IOException e ) { ... }
catch (ClassNotFoundException e2 ) { ... }
```

Сначала наш сервер создает объект сокет `ServerSocket`, прикрепленный к порту 1234. В некоторых системах существуют правила относительно того, какие порты доступны приложениям. Номера портов ниже 1024 обычно отводятся для системных процессов и стандартных, *известных* сервисов, поэтому мы выбрали номер за пределами этого диапазона. Единственная копия объекта `ServerSocket` позволяет принимать любое количество входящих подключений.

Дальше мы входим в цикл, ожидая, когда метод `accept()` из объекта `ServerSocket` вернет активное соединение с клиентом в виде сокета `Socket`. Установив соединение, мы выполняем серверную часть нашего диалога, закрываем соединение и возвращаемся в начало цикла, чтобы иметь возможность принять очередное подключение. В конце, когда серверное приложение хочет полностью прекратить ожидание новых соединений, вызывается метод `close()` из объекта `ServerSocket`.

Это пример однопоточного сервера; он способен работать одновременно только с одним соединением, и пока эта работа не закончится, он не может вызвать метод `accept()`, чтобы продолжить отслеживать новые подключения. Настоящий сервер должен содержать цикл, который принимает параллельные подключения и передает их на обработку в отдельные потоки выполнения. Реализация многопоточных серверных приложений — это очень обширная тема. Позже в этой главе мы создадим два веб-сервера: первый, совсем крошечный, запускает новый поток для каждого соединения, а второй, более продвинутый, обслуживает большое количество соединений с использованием всего нескольких потоков (благодаря пакету NIO).

Сокеты и безопасность

В предыдущих примерах мы исходили из того, что клиенту позволено подключаться к серверу и что у сервера есть права на прослушивание заданного сокета. Для обычного полноценного приложения это нормальная ситуация (если вас не интересуют другие варианты, можете

пропустить этот раздел). Тем не менее программы, несущие в себе определенный риск (такие как Java-апплеты внутри браузера), выполняются под присмотром системы безопасности, которая может накладывать различные ограничения относительно того, к каким серверам можно или нельзя подключаться, разрешено ли ожидать входящие соединения и т. д.

К примеру, политика безопасности, которая действует в большинстве веб-браузеров, позволяет неподписанным апплетам открывать соединение только с тем сервером, на котором они сами выполняются; то есть, они могут общаться только с тем компьютером, с которого были загружены их *class*-файлы. Неподписанным апплетам обычно не позволяется открывать серверные сокеты для приема входящих подключений. Хотя они могут обойти это ограничение, используя свой сервер как промежуточное звено. Это позволит таким апплетам связываться с любым компонентом в любой точке сети.

Благодаря политике безопасности вредоносные апплеты лишены возможности сканировать корпоративную сеть и подключаться к доверенным сервисам. Соблюдением безопасности занимается сервер апплета, а не клиентский компьютер. Ограничение доступа к этому серверу снижает потенциальную угрозу от так называемых троянских приложений, которые делают разные пакости на клиентской стороне (вам вряд ли захочется, чтобы ваш прокси-сервер рассылал спам, ведь виноватым окажетесь именно вы).

Если вы собираетесь запускать свое приложение с использованием встроенного диспетчера безопасности (Security Manager), вы должны понимать, что по умолчанию вам будет запрещен любой доступ к сети. Поэтому, чтобы иметь возможность выполнять сетевые подключения, вам необходимо изменить политику безопасности и снабдить свой код соответствующими правами доступа (см. главу 3). Ниже представлен фрагмент файла, который регулирует политику безопасности приложения; в нем указаны полномочия для использования сокетов, позволяющие подключаться к серверу и принимать подключения от других клиентов с использованием портов, которые не требуют особых привилегий:

```
grant {
   permission java.net.SocketPermission
      "*:1024-", "listen,accept,connect";
};
```

Этот файл можно использовать при запуске Java-машины, устанавливая параметры диспетчера безопасности (назовите его **mysecurity.policy**):

```
% java -Djava.security.manager ↵
\-Djava.security.policy=mysecurity.policy MyApplication
```

Клиентское приложение `DateAtHost`

Когда-то в сети было множество компьютеров, которые выдавали свое локальное время через хорошо известный порт. Это были предшественники NTP — сетевого протокола для синхронизации времени (Network Time Protocol). В следующем примере под названием `DateAtHost` производная класса `java.util.Date` вместо значения локальных часов использует время, полученное от удаленного сервера (развернутое обсуждение класса `Date` можно найти в главе 11).

Клиент `DateAtHost` подключается к сервису времени (порт 37) и считывает четыре байта, которые представляют время на удаленном компьютере. Эти четыре байта записаны в необычном формате, который нам придется расшифровать, чтобы получить время. Вот код клиента:

```
//файл: DateAtHost.java
import java.net.Socket;
import java.io.*;
public class DateAtHost extends java.util.Date {
   static int timePort = 37;
   // количество секунд, которое успело пройти с начала
   // 20 века до полночи 1 января 1970 года по Гринвичу
   static final long offset = 2208988800L;
   public DateAtHost( String host ) throws IOException {
      this( host, timePort );
   }
   public DateAtHost( String host, int port ) throws IOException {
      Socket server = new Socket( host, port );
      DataInputStream din =
new DataInputStream( server.getInputStream( ) );
      int time = din.readInt( );
      server.close( );
      setTime( (((1L << 32) + time) - offset) * 1000 );
   }
}
```

Вот, собственно, и все. Даже несмотря на некоторые излишества, пример получился довольно компактным. Мы предоставили для класса `DateAtHost` сразу два конструктора. Первый принимает только имя сервера и используется в большинстве ситуаций. Второму нужно дополнительно передать номер порта, на котором работает удаленный сервис времени (это необходимо в тех случаях, когда сервис работает на нестандартном порту). Всю основную работу по созданию подключения и установлению времени выполняет второй конструктор, а первый просто вызывает его (используя конструкцию `this()`), передавая в качестве аргумента номер порта по умолчанию. Предоставление упрощенных конструкторов, внутри которых вызываются их более продвинутые аналоги со стандартными параметрами, является для языка Java широко распространенным и довольно полезным приемом (именно поэтому мы его здесь и продемонстрировали).

Второй конструктор подключается к указанному порту удаленного сервера. В качестве обертки для входящего потока он создает объект `DataInputStream`, и затем считывает целое 4-байтовое число при помощи метода `readInt()`. В итоге мы получаем байты в правильном порядке, и это не совпадение. Классы `DataInputStream` и `DataOutputStream` работают с целыми типами, соблюдая *сетевой порядок байтов* (от старшего к младшему). Протокол обмена временем (как и другие стандартные сетевые протоколы, которые работают с двоичными данными) тоже использует этот стандарт, поэтому нам не нужно выполнять никаких преобразований. Конвертирование данных могло бы понадобиться, если бы мы применяли нестандартный протокол, особенно при работе с клиентами или серверами, написанными на каком-то другом языке. В этом случае нам бы пришлось считывать информацию побайтно и делать перестановки на ходу, чтобы получить нужное четырехбайтовое значение. Прочитав данные, мы заканчиваем работу с сокетом и закрываем его, разрывая соединение с сервером. В конце конструктор инициализирует оставшуюся часть объекта; для этого он вызывает метод `setTime()` из класса `Date`, передавая ему полученное значение.

Четыре байта, составляющие значение времени, интерпретируются как целое число, которое описывает количество секунд, прошедших с начала 20 века. Наш класс `DateAtHost` преобразует его в формат абсолютного времени, принятый в языке Java — то есть количество миллисекунд, прошедших с 1 января 1970 года (стандартная дата для языка C и систем семейства Unix). Сначала мы создаем переменную типа long, которая является беззнаковым эквивалентом целого числа, представляющего время. Затем мы вычитаем его из нашего значения, чтобы время отсчитывалось

относительно эры Unix (1 января 1970 года), а не начала XX века. Полученное число умножается на 1000, чтобы преобразовать секунды в миллисекунды. Итоговое значение используется для инициализации объекта.

Внешне классы `DateAtHost` и `Date` почти ничем не отличаются, хотя первый запрашивает время у удаленного сервера, а второй использует локальное время текущего компьютера. Нужно только иметь в виду, что конструктор `DateAtHost` может сгенерировать исключение `IOException`:

```
try {
   Date d = new DateAtHost( "someserver.net" );
   System.out.println( "Время на удаленном сервере: " + d );
}
catch ( IOException e ) { ... }
```

Этот пример выводит значение времени, которое возвращает сервер **someserver.net**.

Серверное приложение TinyHttpd

Хотелось ли вам когда-нибудь написать свой собственный веб-сервер? Что ж, тогда считайте, что вам повезло. В этом разделе мы создадим минималистичный но вполне рабочий веб-сервер под названием TinyHttpd, который прослушивает заданный порт и обслуживает простые HTTP-запросы типа GET. Эти запросы представляют собой обычные текстовые команды, которые выглядят следующим образом:

```
GET /путь/имя_файла [ необязательные параметры ]
```

Подобные команды ваш веб-браузер отправляет при открытии любой страницы в сети. Помимо чтения самого запроса наш сервер пытается открыть указанный файл и вернуть его содержимое. Если страница содержит ссылки на картинки или другие вложенные элементы, браузер делает дополнительные запросы типа GET. Чтобы повысить производительность, сервер TinyHttpd обслуживает каждый запрос в отдельном потоке выполнения. Таким образом, он может обрабатывать запросы параллельно.

Несмотря на свою работоспособность, этот пример слишком уж упрощен. Прежде всего, он реализует очень старое подмножество протокола HTTP, что может не очень понравиться некоторым веб-браузерам. (Во время написания этой книги код проверялся с помощью веб-браузера Safari в операционной системе OS X; со своей задачей он справлялся до-

статочно неплохо.) Также нужно помнить о том, что путь к файлу на платформе Java может в некоторой степени зависеть от архитектуры. Этот пример должен одинаково выполняться на большинстве систем, но для надежности его стоит немного доработать. Мы можем воспользоваться информацией о среде выполнения, которую предоставляет платформа Java, чтобы дополнить наш код и адаптировать его к работе в рамках локальной системы (о том, как это делается, можно узнать в главе 12).



Если у вас нет брандмауэра или других средств обеспечения безопасности, этот пример будет работать на вашем компьютере без всякой защиты. Не применяйте его для решения реальных задач!

Итак, чтобы дальше не затягивать, вот код сервера TinyHttpd:

```
//файл: TinyHttpd.java
import java.net.*;
import java.io.*;
import java.util.regex.*;
import java.util.concurrent.*;
public class TinyHttpd {
   public static void main( String argv[] ) throws IOException {
      Executor executor = Executors.newFixedThreadPool(3);
      ServerSocket ss = new ServerSocket( Integer.parseInt(argv[0]) );
      while ( true )
         executor.execute( new TinyHttpdConnection( ss.accept() ) );
   }
}
class TinyHttpdConnection implements Runnable {
   Socket client;
   TinyHttpdConnection ( Socket client ) throws SocketException {
      this.client = client;
   }
   public void run() {
      try {
         BufferedReader in = new BufferedReader(
            new InputStreamReader(client.getInputStream(), "8859_1" ) );
         OutputStream out = client.getOutputStream();
         PrintWriter pout = new PrintWriter(
            new OutputStreamWriter(out, "8859_1"), true );
```

```
            String request = in.readLine();
            System.out.println( "Запрос: "+request);
            Matcher get = Pattern.compile("GET /?(\\S*).*").matcher( request
);
            if ( get.matches() ) {
               request = get.group(1);
               if ( request.endsWith("/") || request.equals("") )
                  request = request + "index.html";
               try {
                  FileInputStream fis = new FileInputStream ( request );
                  byte [] data = new byte [ 64*1024 ];
                  for(int read; (read = fis.read( data )) > -1; )
                     out.write( data, 0, read );
                  out.flush();
               } catch ( FileNotFoundException e ) {
                  pout.println( "404 Объект не найден" ); }
            } else
               pout.println( "400 Неверный запрос" );
            client.close();
         } catch ( IOException e ) {
            System.out.println( "Ошибка ввода/вывода " + e ); }
      }
}
```

Скомпилируйте класс `TinyHttpd` и поместите его в свою директорию classpath, как было показано в главе 3. Перейдите в каталог с какими-нибудь интересными документами и запустите сервер, указав в качестве аргумента свободный порт. Например:

```
% java TinyHttpd 1234
```

Теперь у вас должна появиться возможность извлекать файлы со своего компьютера с помощью веб-браузера. В URL-адресе нужно указывать номер порта, который вы выбрали. Например, если сервер называется **foo.bar.com** и вы запустили TinyHttpd так, как показано выше, то путь к файлу будет выглядеть следующим образом:

http://*foo.bar.com:1234/welcome.html*

Или, если ваши сервер и веб-браузер работают на одном компьютере (что более вероятно), адрес будет выглядеть так:

http://*localhost:1234/welcome.html*

Программа TinyHttpd ищет файлы относительно своей текущей директории, и именно с нее должны начинаться пути к файлам, которые вы указываете. (Уже успели открыть какие-нибудь документы? Заметили, что при открытии HTML-файлов ваш веб-браузер автоматически генерирует дополнительные запросы для различных элементов веб-страницы, таких как изображения?) Давайте рассмотрим этот процесс подробней.

Сервер TinyHttpd состоит из двух компонентов. Публичный класс `TinyHttpd` содержит главный метод нашего приложения, `main()`. Данный метод начинается с создания сокета `ServerSocket`, привязанного к заданному порту. Затем выполняется цикл, внутри которого происходит ожидание подключения со стороны клиента; в цикле объявляется второй класс, `TinyHttpdConnection`, который и обслуживает конкретный запрос. В каждой итерации мы ждем, когда метод `accept()` из объекта `ServerSocket` вернет новый клиентский сокет `Socket`. Полученный сокет передается в качестве аргумента конструктору класса `TinyHttpdConnection`. Для обслуживания всех наших соединений мы используем объект `Executor` с фиксированным размером пула.

Класс `TinyHttpdConnection` реализует интерфейс `Runnable`. Для каждого соединения мы запускаем поток, который работает достаточно долго для того, чтобы успеть выполнить клиентский запрос и завершиться. Вся магия происходит в теле метода `run()` из класса `TinyHttpdConnection`. Сначала мы извлекаем поток `OutputStream` для передачи данных клиенту. Во второй строчке мы считываем запрос типа GET из потока `InputStream` в переменную `request`. Этот запрос представляет собой объект `String`, который может содержать символы переноса строки; он выглядит так, как мы описывали выше. Чтобы извлечь его содержимое, мы используем буфер `BufferedInputStream` в качестве обертки для объекта `InputStreamReader` (класс `InputStreamReader` мы обсудим в следующем подразделе).

Затем мы приступаем к разбору запроса, чтобы извлечь имя файла. Здесь используется API-интерфейс для работы с регулярными выражениями (всю информацию о регулярных выражениях и API-интерфейсе для работы с ними вы найдете в главе 10). Наше выражение ищет подстроку «GET», за которой может следовать слеш и набор любых печатных символов. В конце мы добавляем шаблон «*.», чтобы выражение охватывало весь ввод; так мы сможем проверять весь запрос целиком, используя метод `match()` из объекта `Matcher`. Та часть выражения, которая соответствует имени файла, находится в группе «(\\S*)». Таким образом, мы можем извлечь ее с помощью метода `group()`. В конце мы

проверяем, похоже ли имя файла на директорию (то есть, заканчивается ли оно слешем), и не является ли оно пустым. В этом случае, чтобы пользователю было удобней, мы добавляем к пути уже знакомое нам стандартное имя файла — **index.html**.

Получив имя файла, мы пытаемся его открыть и отправить его содержимое в виде большого массива байтов. Внутри цикла мы последовательно считываем буферы и передаем их клиенту через поток `OutputStream`. Если запрос нельзя разобрать или указанного файла не существует, мы используем объект `PrintStream`, чтобы отправить текстовые данные, после чего возвращаем стандартное для протокола HTTP сообщение об ошибке. В конце мы закрываем сокет и выходим из метода `run()`, завершая тем самым нашу задачу.

Разговаривают ли российские веб-серверы по-русски?

В вышеописанном примере мы специально создали объекты `InputStreamReader` и `OutputStreamWriter` для буфера `BufferedRead` и экземпляра `PrintWriter` соответственно. Благодаря этому можно указать кодировку, которая будет использоваться для преобразования сообщений HTTP-протокола в байты и наоборот. (Обратите внимание: речь идет не о содержимом отправляемого нами файла — для нас это просто поток ничего незначащих байтов; мы говорим о запросах типа GET и ответных сообщениях.) По умолчанию устанавливается стандартная системная кодировка. Она подходит в большинстве случаев, но здесь мы имеем дело со строгим международным протоколом, поэтому нам нужно быть более конкретными. В спецификации RFC для HTTP-протокола сказано, что клиенты и серверы должны кодировать символы в формате ISO8859-1. Мы специально указали эту кодировку при создании объектов `InputStreamReader` и `OutputStreamWriter`. Оказывается, ISO8859-1 — это всего лишь еще одно название знакомого нам стандарта ASCII; во время преобразования символов в кодировку Unicode и обратно все значения из набора ASCII остаются неизменными, поэтому даже со стандартной кодировкой у нас могло бы не возникнуть никаких проблем. Но о таких вещах все же стоит иногда вспоминать, что мы и сделали.

Укрощение сервера

У сервера TinyHttpd есть одна важная проблема: он не ограничен в доступе к файлам. Проявив немного смекалки, мы можем заставить его вернуть клиенту любой файл, который находится на вашем компью-

тере. Было бы неплохо запретить серверу TinyHttpd обслуживать файлы за пределами рабочей директории или какого-то подкаталога. Проще всего этого можно добиться с помощью стандартного для платформы Java диспетчера безопасности. Обычно этот способ используется для того, чтобы код, загруженный по сети, не мог сделать ничего подозрительного. Но с тем же успехом мы можем заставить диспетчер безопасности ограничивать доступ к файлам в нашем приложении.

Вы можете использовать простейшую политику безопасности — например ту, что была описана ранее в этой главе; она позволит серверу принимать подключения на портах определенного диапазона. По чистой случайности стандартные права доступа к файлам, которые по умолчанию обеспечивает политика безопасности, полностью подходят для нашего примера: они позволяют работать с файлами в текущей рабочей директории и всех ее подкаталогах. Таким образом, чтобы получить именно ту защиту, которая нам нужна в этом конкретном случае, достаточно просто активировать диспетчер безопасности. (Если нужно будет разрешить серверу работать с другими директориями, вы сможете легко указать дополнительные полномочия.)

С активированным диспетчером безопасности сервер не сможет выйти за пределы текущей директории и ее подкаталогов. Но если он попытается это сделать, будет сгенерировано исключение, а доступ к файлу будет закрыт. На этот случай мы должны перехватывать исключение `SecurityException` и возвращать веб-браузеру соответствующее сообщение. Вставьте следующий код сразу после блока `catch` для типа `FileNotFoundException`:

```
...
} catch ( Security Exception e ) {
   pout.println("403 Доступ запрещен");
}
```

Потенциал для улучшения

Сервер TinyHttpd все еще имеет большой потенциал для улучшения. Формально он реализует устаревшую версию протокола HTTP (0.9), в рамках которой сервер должен принимать только запросы типа GET, и возвращать только содержимое файлов. Все современные серверы общаются по протоколу HTTP 1.1, который предусматривает дополнительные метаданные как в запросах, так и в ответах, и требует наличия определенной информации, такой как номер версии и размер содержи-

мого. Вы можете достаточно легко расширить функциональность нашего простого сервера, позволив ему отправлять несколько лишних битов в качестве заголовков. Но этим возможности протокола HTTP 1.1 не ограничиваются; например, он позволяет принимать запросы нескольких клиентов через один сокет, что требует более изощренного кода.

Естественно, настоящие веб-серверы умеют делать много разных вещей. К примеру, за счет нескольких дополнительных строчек кода можно реализовать такую распространенную функцию, как вывод содержимого директории в виде HTML-страницы. Дорабатывая этот пример, вы сможете научиться некоторым интересным приемам.

Свойства сокетов

Как уже было сказано выше, для работы с сокетами в Java используется упрощенный API-интерфейс. В других языках, где программисту доступны все мельчайшие подробности сетевого взаимодействия, сокетам можно передавать множество сложных и неявных параметров, влияя тем самым на поведение низкоуровневых протоколов. Некоторые важные параметры доступны и на платформе Java.

Опция `SO_TIMEOUT`

Опция `SO_TIMEOUT` устанавливает таймер для всех блокирующих операций ввода/вывода, которые выполняет сокет, чтобы вам не пришлось бесконечно ждать их завершения. Она работает во всех сокетах и доступна для таких методов, как `accept()` (только на серверной стороне), `read()` или `write()`. Если таймер завершает отсчет до момента окончания операции, генерируется исключение `InterruptedIOException`. Обработав исключение, вы можете прервать операцию или продолжать использовать сокет в обычном режиме. Многопоточные блокирующие серверы, такие как TinyHttpd, могут применять подобный прием для завершения своей работы:

```
serverSocket.setSoTimeout( 2000 ); // 2 секунды
while ( !shutdown ) {
   try {
      Socket client = serverSocket.accept();
      handleClient( client );
   } catch ( InterruptedIOException e ) {
      // игнорируем исключение
```

```
    }
    // выход
}
```

Таймер устанавливается с помощью метода `setSoTimeout()` из класса `Socket`; в качестве аргумента выступает время ожидания (в миллисекундах). Это актуально как для обычных сокетов, так и для классов `ServerSocket` (TCP) и `DatagramSocket` (UDP), рассмотренных ранее. Чтобы получить время ожидания, используйте метод `getSoTimeout()`.

Блокирование потоковых операций ввода/вывода и невозможность проверить их активность — это серьезная проблема для платформы Java, и использование таймеров решает ее только частично. Позже в этой главе мы закончим рассмотрение пакета NIO, который предоставляет полный набор неблокирующих операций ввода/вывода всех видов, включая сокеты.

Опция `TCP_NODELAY`

Эта опция отключает в протоколе TCP функцию под названием «алгоритм Нейгла», призванную предотвращать засорение сети мелкими пакетами, которые отправляют определенные интерактивные приложения. Например, в своей первой сетевой программе мы отправляли по одному байту на каждую операцию записи. Если бы мы включили эту опцию, реализация протокола TCP в определенных случаях могла бы задерживать отправку байта на какое-то очень короткое время, дожидаясь поступления следующего пакета данных. Если ваша сеть достаточно быстрая, вы можете отключить эту «задержку», чтобы все пакеты отправлялись как можно быстрее. Метод `setTcpNoDelay()` из класса `Socket` принимает булево значение, которое определяет наличие задержки. Чтобы узнать, активирована ли опция `TCP_NODELAY`, вызовите метод `getTcpNoDelay()`, который тоже возвращает значение типа `boolean`.

Опция `SO_LINGER`

Эта опция определяет, что нужно делать с неотправленными данными, когда для активного соединения вызывается метод `close()`. Обычно система блокирует закрытие до тех пор, пока все данные из буфера не будут доставлены. Метод `setSoLinger()` из класса `Socket` принимает два аргумента: значение типа `boolean`, которое включает или выключает данную опцию, и целое число, которое определяет время ожидания (в секундах). Если время ожидания равно 0, все данные, которые не успели

отправиться, будут проигнорированы, а соединение по протоколу TCP будет разорвано (завершено и сброшено). Чтобы узнать текущее время ожидания, вызовите метод `getSoLinger()`.

Опция `TCP_KEEPALIVE`

Эту опцию можно включить с помощью метода `setKeepAlive()`. Она отвечает за функцию протокола TCP, которая в случае отсутствия активности каждые два часа опрашивает компьютер по другую сторону соединения. Обычно TCP-соединение не отправляет никаких пакетов, если данные не передаются. Поэтому сложно сказать, что происходит с компьютером на противоположном конце соединения: может, он куда-то пропал, а может, ему просто нечего передавать. Корректное отключение любой из сторон всегда можно определить. Но если один из компьютеров куда-то исчез, мы не узнаем об этом до тех пор, пока не попробуем с ним пообщаться. В связи с этим серверы часто используют опцию `TCP_KEEPALIVE`, чтобы узнавать об утерянных соединениях с клиентами (в противном случае они могли бы только отвечать на запросы, а не инициировать их). Данная опция не входит в описание протокола TCP; это всего лишь дополнение, наличие которого не гарантируется. Лучшим решением будет реализовать ее самостоятельно на уровне своего протокола.

Полуоткрытое TCP-соединение

Протокол TCP позволяет закрывать одно из направлений потока. Проще говоря, вы можете запретить отправку пакетов, но оставить возможность их получения (и наоборот). Некоторые протоколы используют эту возможность для оповещения об окончании запроса, закрывая соединение на клиентской стороне, чтобы сервер мог определить завершение потока. Вы можете закрыть сокет в любом направлении, используя методы `shutdownOutput()` и `shutdownInput()`.

Прокси и брандмауэры

Большая часть современных сетей защищена брандмауэрами. Некоторые брандмауэры не только предотвращают вторжение извне, но и не дают внутренним приложениям использовать сокеты для непосредственного подключения к внешним сетям. Часто в таких случаях внутрисетевым клиентам предоставляется сервис под названием SOCKS (сокращенно от «sockets» — сокеты), который играет роль прокси-

сервера для соединений на уровне сокетов; это помогает администраторам следить за тем, какие соединения разрешены. У брандмауэра могут быть предусмотрены прокси-серверы для отдельных высокоуровневых протоколов, таких как HTTP и FTP, что дает еще больший контроль над происходящим в сети и позволяет просматривать передаваемые данные. Все это делается для того, чтобы знать, кто, к кому и для чего подключается.

Но если ваш брандмауэр запрещает вам выходить за пределы сети даже через сервис SOCKS, вы все равно можете общаться с внешним миром, передавая и получая данные по протоколу HTTP. О том, как выполнять запросы типа POST и GET, вы можете узнать в главе 14.

Платформа Java имеет встроенную поддержку сервисов SOCKS, а также прокси-серверов для протоколов HTTP и FTP. От вас только требуется указать для своего приложения некоторые параметры (в случае с апплетами у вас не будет доступа к этим параметрам — всю необходимую конфигурацию предоставляет веб-браузер). Чтобы настроить платформу Java для работы с сервисом SOCKS (версий 4 или 5), установите следующие системные свойства:

`socksProxyHost`

Имя сервиса SOCKS.

`socksProxyPort`

Номер порта сервиса SOCKS.

При необходимости вы можете предоставить логин и пароль для сервиса SOCKS, указав дополнительные свойства `java.net.socks.username` и `java.net.socks.password`.

Схожим образом обеспечивается поддержка прокси-серверов для протоколов HTTP и FTP (свойства устанавливаются отдельно для каждого протокола):

`http.proxyHost`
`ftp.proxyHost`

Имя прокси-сервера

`http.proxyPort`

Номер порта прокси-сервера

`http.nonProxyHosts`

Список серверов, к которым всегда можно подключаться напрямую. Элементы списка разделяются вертикальной чертой и могут содержать звездочку (*) в качестве группового символа — например, `myserver|*.mydomain`.

Вы можете указать все эти свойства с помощью метода `System.setProperty()` или в командной строке, запустив Java-машину с опцией «`-D`». Следующая команда запускает программу MyProgram с использованием прокси-сервера, который имеет адрес **foo.bar.com**, привязан к порту 1234 и предоставляет доступ к протоколу HTTP:

```
% java -Dhttp.proxyServer=foo.bar.com -Dhttp.proxyPort=1234 MyProgram
```

Класс `ProxySelector`

Платформа Java также предоставляет API-интерфейс для внутрипрограммного контроля за использованием прокси-серверов. Класс `java.net.ProxySelector` содержит метод, который принимает идентификатор URI (частным случаем которого является URL-адрес; см. главу 14), и возвращает список объектов `java.net.Proxy`. Эти объекты представляют прокси-серверы или прямые соединения, которые используются для заданного протокола. По умолчанию объект `ProxySelector` действует в соответствии с системными свойствами, которые были рассмотрены выше. При необходимости вы можете создать и установить свой собственный экземпляр `ProxySelector`, чтобы управлять процессом подключения и проводить разные соединения через разные прокси-серверы. Чтобы узнать текущие настройки, вы можете получить стандартный объект `ProxySelector`, воспользовавшись статическим методом `ProxySelector.getDefault()`, и просмотреть состояние различных протоколов с помощью метода `select()`. Следующий код выводит текстовую информацию о предпочтительных прокси-серверах (если таковые имеются) для конкретного URL-адреса:

```
ProxySelector ps = java.net.ProxySelector.getDefault();
List list = ps.select( new URI("http://java.sun.com/") );
System.out.println( list.get(0) ); //то есть, HTTP@myserver:1234
```

Объект `Proxy` содержит подробную информацию, включая тип указанного идентификатора (`DIRECT`, `HTTP` или `SOCKS`) и адрес прокси-сервера. Чтобы получить сведения о сервисе SOCKS для заданного сервера и порта, используйте идентификатор URI вида «`socket://сервер:порт`».

Датаграммные сокеты

Для соединения с клиентом сервер TinyHttpd использовал объект `Socket`. В том примере целостность данных обеспечивалась на уровне протокола TCP; нам не нужно было беспокоиться о том, что отправленные нами байты будут получены не в том порядке или с какими-то ошибками. Теперь давайте избавимся от этой страховки и напишем аплет на основе класса `java.net.DatagramSocket`, в котором используется протокол UDP. Датаграмма — это что-то вроде записи, отправляемой через почтовое отделение; это отдельный фрагмент данных, который передается в виде единого пакета.

В предыдущем примере мы могли получить из нашего сокета удобный исходящий поток `OutputStream`, и записывать данные в него, словно это файл; в случае с объектом `DatagramSocket` нам придется работать отдельно с каждой датаграммой. (Конечно, протокол TCP тоже «нарезает» наши данные на пакеты, но это происходит автоматически, без нашего участия.)

Протокол UDP не гарантирует получения отправленных данных. Но даже если пакеты дойдут до адресата, их порядок может быть нарушен; к тому же датаграммы могут дублироваться (в очень редких случаях). Представьте, что вы вырываете из энциклопедии отдельные страницы, засовываете их в конверты и отправляете почтой своему другу — вот так и работает протокол UDP. Если ваш друг захочет прочитать энциклопедию, ему придется самостоятельно упорядочивать все страницы. Если часть конвертов затерялась на почте, ваш друг должен будет отправить вам письмо с просьбой выслать замену.

Естественно, было бы неразумно передавать по UDP большие объемы данных, не исправляя ошибки. Тем не менее по своей эффективности этот протокол значительно превосходит TCP, особенно если для вас неважны порядок получения сообщений и стопроцентная гарантия их доставки. Возьмем, к примеру, базу данных: клиент периодически отправляет простые запросы, подтвердить получение которых сервер может самим фактом ответа. Если ответ не приходит в течение определенного времени, клиент может отправить еще один запрос. У клиента не должно возникнуть особых проблем при сопоставлении запросов с ответами. Протокол UDP используется в таких важных программных продуктах как DNS (Domain Name System — система доменных имен) и NFS (Network File System — сетевая файловая система от компании Sun).

Апплет HeartBeat

В этом разделе мы создадим простой апплет под названием HeartBeat, который работает внутри веб-браузера; при запуске и остановке он будет отправлять на сервер датаграммы. Мы также напишем простое, но полноценное серверное приложение, Pulse, которое будет получать эти датаграммы и выводить их на экран. Вывод сервера даст вам примерное представление о том, кто и когда просматривает вашу веб-страницу (мы исходим из того, что брандмауэр не блокирует UDP-пакеты). Это та задача, для которой протокол UDP подходит как нельзя лучше: нам не нужны накладные расходы, обусловленные использованием TCP-сокетов, и если датаграмма потеряется, ничего страшного не произойдет.

Для начала приведем код апплета HeartBeat:

```
//файл: HeartBeat.java
import java.net.*;
import java.io.*;
public class HeartBeat extends java.applet.Applet {
String myHost;
   int myPort;
   public void init( ) {
      myHost = getCodeBase().getHost( );
      myPort = Integer.parseInt( getParameter("myPort") );
   }
   private void sendMessage( String message ) {
      try {
         byte [] data = message.getBytes("UTF-8");
         InetAddress addr = InetAddress.getByName( myHost );
         DatagramPacket pack =
            new DatagramPacket( data, data.length, addr, myPort );
         DatagramSocket ds = new DatagramSocket();
         ds.send( pack );
         ds.close();
      } catch ( IOException e ) {
         System.out.println( e ); // Ошибка при создании сокета
      }
}
public void start( ) {
   sendMessage("Запустился");
}
```

```
    public void stop( ) {
        sendMessage("Остановился");
    }
}
```

Скомпилируйте апплет и вставьте его в HTML-документ с помощью тега <applet>:

```
<html><body>
    <h1>Heartbeat!</h1>
    <applet height="1 " width="1 " code="HeartBeat ">
        <param name="myPort" value="1234">
    </applet>
</body></html>
```

Убедитесь в том, что скомпилированный файл **HeartBeat.class** находится в одной директории с HTML-документом (который мы назовем **heartbeat.html**). Подробней о встраивании апплетов в веб-страницы мы поговорим в главе 23.

Параметр `myPort` должен обозначать номер порта, на котором серверное приложение ожидает появления данных (мы выбрали «1234»).

Теперь приступим к серверу Pulse:

```
//файл: Pulse.java
import java.net.*;
import java.io.*;
public class Pulse {
    public static void main( String [] argv ) throws IOException {
DatagramSocket s =
new DatagramSocket( Integer.parseInt(argv[0]) );
        while ( true ) {
            DatagramPacket packet =
new DatagramPacket( new byte [1024], 1024 );
            s.receive( packet );
            System.out.println( "длина пакета = "+packet.getData().length );
            String message = new String( packet.getData(), "UTF-8" );
            System.out.println( "Сообщение от: "
+ packet.getAddress().getHostName()
+ " - " + message );
        }
    }
}
```

Скомпилируйте этот код и запустите его на своем веб-сервере, указав в качестве аргумента номер порта:

```
% java Pulse 1234
```

Номер порта должен совпадать с параметром `myPort`, который вы указали в теге `<applet>` для приложения HeartBeat.

Теперь, чтобы запустить наш пример, нам понадобится веб-сервер. Мы не можем просто открыть файл в веб-браузере, потому что, как мы уже упоминали ранее, апплетам разрешено общаться только с тем компьютером, на котором они запущены; без наличия веб-сервера диспетчер безопасности будет блокировать любое сетевое взаимодействие. К счастью, в начале этой главы мы уже написали вполне достойный веб-сервер — TinyHttpd! Просто запустите его в одной директории с файлом **heartbeat.html** (проследите за тем, чтобы выбранный вами номер порта не совпал с тем, который используется в клиенте HeartBeat и сервере Pulse), и откройте через него свою страницу.

Мы проверяли данный пример с помощью веб-браузера Safari в операционной системе Mac OS X, и на момент написания этой книги он работал так, как мы того ожидали. Но если у вас возникли какие-то проблемы, вы можете попробовать «настоящий» веб-сервер — мы не возражаем.

Теперь откройте эту веб-страницу в своем веб-браузере. Вы не увидите на ней ничего особенного (более продуманное приложение могло бы как-то визуализировать свою работу), но сервер Pulse должен проявить кое-какую активность. При каждом обновлении страницы наш апплет останавливается и запускается снова, отправляя сообщения, которые выводит приложение Pulse:

```
Сообщение от: foo.bar.com - Запустился
Сообщение от: foo.bar.com - Остановился
Сообщение от: foo.bar.com - Запустился
Сообщение от: foo.bar.com - Остановился
Сообщение от: foo.bar.com - Запустился
Сообщение от: foo.bar.com - Остановился
...
```

Круто, правда? Только не забывайте, что доставку датаграмм никто не гарантирует (хотя возникновение такого сбоя в нормальной сети очень маловероятно), и вполне возможно, что одно из сообщений будет пропущено. Теперь давайте возьмемся за код.

Код апплета HeartBeat

Апплет `HeartBeat` переопределяет методы `init()`, `start()` и `stop()` из класса `Applet`, реализуя при этом свой собственный частный метод `sendMessage()`, который шлет датаграммы. (Мы пока еще не приступали к изучению апплетов, поэтому, если вы хотите узнать больше подробностей, можете перейти к главе 23.) Работа апплета начинается с метода `init()`, где определяется адресат будущих сообщений. Мы использовали методы `getCodeBase()` и `getHost()`, чтобы узнать имя текущего компьютера и получить подходящий номер порта, который был указан в параметре `myPort` внутри тега `<applet>`. Когда апплет завершит свою инициализацию, его запуск и остановка будут приводить к вызовам методов `start()` и `stop()`, каждый из которых отправляет соответствующее сообщение с помощью метода `sendMessage()`.

Метод `sendMessage()` отвечает за передачу на сервер текстовых сообщений в виде датаграмм. Он принимает объект `String`, делает из него пакет с сообщением, и отправляет. Вся информация о датаграмме, включая имя сервера-получателя и номер порта, хранится в объекте `java.net.DatagramPacket`. Это как конверт с обратным адресом, напичканный нашими байтами. Создав объект `DatagramPacket`, метод `sendMessage()` отправляет его через свежеоткрытый сокет `DatagramSocket`.

На создание пакета отводится первых пять строчек метода `sendMessage()`:

```
try {
   byte [] data = message.getBytes("UTF-8");
   InetAddress addr = InetAddress.getByName( myHost );
   DatagramPacket pack =
      new DatagramPacket(data, data.length, addr, myPort );
```

Сначала содержимое строки `message` помещается в массив байтов под названием `data`. Затем в результате поиска адреса сервера `myHost` создается объект `java.net.InetAddress`. Этот объект хранит сетевой IP-адрес, который извлекается с помощью статического метода `getByName()` и системного *сервиса доменных имен*. (Мы еще вернемся к классу `InetAddress` в следующем разделе.) В конце этого фрагмента мы вызываем конструктор `DatagramPacket`, который принимает четыре параметра: массив байтов с нашими данными внутри, размер данных, объект с адресом получателя и номер порта. Строка кодируется в формат UTF-8, который будет использоваться и при декодировании.

В оставшихся строчках мы создаем стандартный клиентский сокет `DatagramSocket` и вызываем из него метод `send()`, чтобы отправить пакет `DatagramPacket`. Сделав все это, мы закрываем сокет:

```
DatagramSocket ds = new DatagramSocket();
ds.send( pack );
ds.close();
```

В этом коде две операции могут сгенерировать исключение типа `IOException`: поиск адреса (`InetAddress.getByName()`) и вызов метода `send()` из объекта `DatagramSocket`. В первом случае может быть сгенерировано исключение `UnknownHostException`, которое является разновидностью `IOException` и сигнализирует о том, что адрес сервера получить не удалось. Если метод `send()` выдаст экземпляр `IOException`, это будет означать, что на клиентской стороне возникли серьезные проблемы с сетью. Мы перехватываем данные ошибки и выводим в блоке `catch` сообщения о том, что случилось нечто непредвиденное. Получив одно из этих исключений, мы можем смело предполагать, что датаграмма не дошла до адресата. Однако эту логику нельзя применить к обратному утверждению: даже если исключения не было, мы не можем знать наверняка, доступен ли сервер, и был ли принят наш пакет. API-интерфейс класса `DatagramSocket` не предоставляет такой информации.

Код сервера Pulse

Сервер Pulse общается с апплетом HeartBeat. Сначала он создает сокет `DatagramSocket`, чтобы отслеживать подключения на заданном порту. На этот раз мы передаем номер порта в конструктор; мы извлекаем его из командной строки в текстовом виде (`argv[0]`) и конвертируем в целое число с помощью метода `Integer.parseInt()`. Обратите внимание на разницу между этим вызовом конструктора и аналогичной операцией в классе `HeartBeat`. Сервер должен отслеживать входящие датаграммы на заранее определенном порту, поэтому порт нужно задать при создании сокета `DatagramSocket`. Клиент же просто отправляет датаграммы, поэтому номер порта можно указать в последний момент, когда создается сам пакет.

После этого, чтобы принять входящую датаграмму, сервер Pulse создает пустой пакет `DatagramPacket` с ограниченным максимальным размером. Эта версия конструктора класса `DatagramPacket` принимает в качестве аргументов массив байтов и длину запроса. В массиве хранится максимальный допустимый объем полученных данных. (На практике

протокол UDP позволяет передавать по Интернету пакеты размером не больше 8 Кб, хотя в локальных сетях эта планка может быть поднята до 64 Кб.) В конце из сокета `DatagramSocket` вызывается метод `receive()`, который ждет появления данных. Когда пакет наконец доходит до адресата, его содержимое кодируется в строку формата UTF-8 и выводится на экран. Количество байт, которые были получены из пакета, определяется с помощью метода `getLength()`.

Как видите, датаграммные сокеты отличаются от обычных чуть большим количеством рутинных операций и выглядят не так опрятно. Здесь платформа Java мало чем помогает, практически полностью повторяя интерфейс сокетов, принятый в операционных системах семейства Unix. Но в целом датаграммы не такие уж и страшные, если их использовать для решения простых задач.

Класс `InetAddress`

Класс `java.net.InetAddress` — это наиболее низкоуровневый API-интерфейс для работы с IP-адресами. Его экземпляры представляют собой отдельные цифровые адреса, которые можно получать из строковых имен, используя средства самого класса `InetAddress`. В большинстве наших сетевых примеров мы идентифицировали удаленные серверы по их именам, но внутри платформа Java все равно вызывала для нас статический метод `InetAddress.getByName()`, сопоставляя имя с физическим IP-адресом. Обычно для этого используется сервис DNS (чтобы повысить эффективность, результаты в нем кешируются). Большинство сетевых API-интерфейсов на платформе Java (таких как сокеты) могут работать как с именем сервера, так и с объектом `InetAddress`. Класс `InetAddress` также позволяет выполнять обратные запросы (получение имени сервера из IP-адреса) и находить главный адрес текущего компьютера (для этого существует статический метод `InetAddress.getLocalHost()`).

Один из наиболее полезных методов класса `InetAddress`, `isReachable()`, пытается использовать протокол ICMP (с которым работает утилита *ping*), чтобы определить, можно ли получить доступ к удаленному серверу по сети. Это стандартный механизм для проверки доступности узлов сети и определения задержки при передаче пакетов. Данный протокол является низкоуровневым (наряду с TCP и UDP) и может не поддерживаться некоторыми системами. Если методу `isReachable()` не удается отправить ICMP-пакеты, он пытается про-

верить доступность удаленного сервера с помощью протокола ECHO (порт 7), который работает поверх TCP. Например:

```
InetAddress server = InetAddress.getByName("myserver");
If ( !server.isReachable( timeout ) ) // миллисекунды
   pageSomeone();
```

Простой протокол для передачи сериализованных объектов

Ранее в этой главе мы демонстрировали гипотетический пример общения между клиентом и сервером, в котором вместе с простыми данными передавался сериализованный объект. На тот момент передача объектов между двумя программами могла показаться вам чем-то незначительным, но в контексте языка Java, который основан на переносимом байт-коде, это имеет огромное значение. В этом разделе мы займемся созданием протокола с использованием сериализованных объектов.

Но прежде чем приступать к делу, стоит рассмотреть сетевые протоколы в целом. Большинство программистов считает работу с сокетами скучной и сложной. Даже несмотря на то, что платформа Java значительно упрощает этот процесс, главная задача сокетов все равно заключается лишь в передаче неструктурированных потоков байтов между двумя точками. Если вы хотите использовать сокеты для серьезного сетевого взаимодействия, первое, что вам нужно сделать — это разработать протокол, который определяет, какие данные вы будете отправлять и получать. Сложней всего описать принцип, по которому информация упаковывается перед передачей по сети и распаковывается на другой стороне соединения.

Как мы уже знаем, для простых типов данных эта проблема решается с помощью классов `DataInputStream` и `DataOuputStream`. Мы можем считывать и записывать числа, строки и другие простые типы в стандартном формате, который понимает любая Java-машина. Но для выполнения реальных задач нам придется объединять эти простые данные в более масштабные структуры. Платформа Java предоставляет механизм сериализации, позволяющий легко и непринужденно передавать наши данные в том состоянии, в котором они потом будут использоваться — в виде обычных объектов. Мы можем упаковывать целые иерархии взаимосвязанных объектов, чтобы позже продолжить работу с ними уже на другой Java-машине.

Простой сервер для обмена объектами

В следующем примере клиент и сервер обмениваются сериализованными данными. Запрос клиента и ответ сервера представлены двумя разными объектами. Сетевое общение заканчивается, когда клиент закрывает соединение. Сложно себе представить более простой протокол. Все сложные детали скрыты за процессом сериализации, благодаря чему мы можем работать со стандартными объектами в привычной для нас манере.

Для начала мы определим общий базовый класс, `Request`, на основе которого будут строиться различные виды запросов к серверу. Это позволит нам легко определять, является ли объект запросом. В реальном приложении мы могли бы хранить в этом классе такую информацию, как имена и пароли клиентов, временные отметки, серийные номера и т. д. В нашем случае класс `Request` может быть пустым и служить всего лишь отправной точкой для разных запросов:

```
//файл: Request.java
public class Request implements java.io.Serializable {}
```

Класс `Request` реализует интерфейс `Serializable`, поэтому все его производные изначально пригодны для сериализации. Теперь давайте создадим конкретные типы запросов. Первый, `DateRequest`, тоже выглядит крайне просто. С его помощью мы будем запрашивать у сервера объект `java.util.Date`:

```
//файл: DateRequest.java
public class DateRequest extends Request {}
```

Следующий на очереди универсальный объект `WorkRequest`. Клиент будет передавать его на сервер, чтобы тот выполнил какие-нибудь вычисления. Происходит следующий процесс: сервер вызывает из объекта `WorkRequest` метод `execute()`, проделывая всю работу на своей стороне, а итоговый результат возвращает клиенту в качестве ответа:

```
//файл: WorkRequest.java
public abstract class WorkRequest extends Request {
   public abstract Object execute();
}
```

Мы создаем в приложении класс `MyCalculation`, который является наследником `WorkRequest` и содержит код для выполнения определенных вычислений (в данном случае это всего лишь возведение числа в квадрат):

```
//файл: MyCalculation.java
public class MyCalculation extends WorkRequest {
   int n;
   public MyCalculation( int n ) {
      this.n = n;
   }
   public Object execute() {
      return new Integer( n * n );
   }
}
```

Объект `MyCalculation` несет внутри себя не так уж много информации — всего лишь одно целое число. Однако с тем же успехом он мог бы содержать большой объем данных, включая ссылки на множество других объектов, которые являются частью сложных структур — например массивов или связных списков. Есть только одно ограничение: передаваемые объекты должны быть доступны для сериализации или помечены ключевым словом `transient` (см. главу 12). Стоит заметить, что объект `MyCalculation` обладает не только состоянием, но и поведением — он содержит операцию `execute()`. И хотя во время сериализации передаются только данные объекта, позже при обсуждении технологии RMI вы увидите, что платформа Java способна динамически загружать байт-код классов, позволяя передавать по сети как поля, так и методы.

Также обратим ваше внимание на то, что даже без использования динамической загрузки классов по сети (что на практике случается не так часто) этот шаблон проектирования имеет большое значение (обычно его так и называют — «команда»). Применяя полиморфизм, чтобы скрыть от сервера подробности поведения объекта, мы получаем возможность легко расширять наше приложение. Полиморфизм и сериализация прекрасно работают в связке.

Теперь, когда у нас есть протокол, пришло время заняться сервером. Класс `Server`, представленный ниже, во многом похож на приложение TinyHttpd, которое мы разрабатывали ранее:

```
//файл: Server.java
import java.net.*;
import java.io.*;
public class Server {
   public static void main( String argv[] ) throws IOException {
      ServerSocket ss = new ServerSocket( Integer.parseInt(argv[0]) );
```

```
      while ( true )
         new ServerConnection( ss.accept() ).start( );
   }
} // на этом класс Server заканчивается
class ServerConnection extends Thread {
   Socket client;
   ServerConnection ( Socket client ) throws SocketException {
      this.client = client;
      setPriority( NORM_PRIORITY - 1 );
   }
   public void run(   ) {
      try {
         ObjectInputStream in =
            new ObjectInputStream( client.getInputStream( ) );
         ObjectOutputStream out =
            new ObjectOutputStream( client.getOutputStream(   ) );
         while ( true ) {
            out.writeObject( processRequest( in.readObject(   ) ) );
            out.flush(   );
         }
      } catch ( EOFException e3 ) { // нормальное завершение потока
         try {
            client.close(   );
         } catch ( IOException e ) { }
      } catch ( IOException e ) {
// ошибка ввода/вывода
         System.out.println( "ошибка ввода/вывода " + e );
      } catch ( ClassNotFoundException e2 ) {
// тип запроса неизвестен
         System.out.println( e2 );
      }
   }
   private Object processRequest( Object request ) {
      if ( request instanceof DateRequest )
         return new java.util.Date(  );
      else if ( request instanceof WorkRequest )
         return ((WorkRequest)request).execute(   );
      else
         return null;
   }
}
```

Объект `Server` обрабатывает каждый запрос в отдельном потоке выполнения. Для каждого соединения метод `run()` создает объекты `ObjectInputStream` и `ObjectOutputStream`, с помощью которых сервер принимает запрос и отправляет ответ. Метод `processRequest()` определяет значение запроса и отвечает на него соответствующим образом. Чтобы узнать тип объекта, представляющего запрос, используется оператор `instanceof`.

Теперь осталось реализовать клиентское приложение, которое выглядит еще проще:

```
//файл: Client.java
import java.net.*;
import java.io.*;
public class Client {
   public static void main( String argv[] ) {
      try {
         Socket server =
            new Socket( argv[0], Integer.parseInt(argv[1]) );
         ObjectOutputStream out =
            new ObjectOutputStream( server.getOutputStream(   ) );
         ObjectInputStream in =
            new ObjectInputStream( server.getInputStream(  ) );
         out.writeObject( new DateRequest( ) );
         out.flush(   );
         System.out.println( in.readObject() );
         out.writeObject( new MyCalculation( 2 ) );
         out.flush(   );
         System.out.println( in.readObject() );
         server.close(   );
      } catch ( IOException e ) {
// ошибка ввода/вывода
         System.out.println( "ошибка ввода/вывода " + e );
      } catch ( ClassNotFoundException e2 ) {
         // тип ответа неизвестен
System.out.println( e2 );
      }
   }
}
```

Как и сервер, клиент создает два потока для обмена объектами. Он поочередно отправляет объекты `DateRequest` и `MyCalculation`, выводит на

экран полученные ответы и закрывает соединение. Как на клиенте, так и на сервере после метода `writeObject()` всегда следует вызов `flush()`. Это позволяет отправить все данные, которые остаются в буфере, благодаря чему ожидание ответа начинается уже после того, как противоположная сторона получит весь запрос. Когда клиент закрывает соединение, сервер перехватывает исключение `EOFException` и завершает сессию. Как вариант, наш клиент мог бы отправлять специальное значение (возможно, `null`), чтобы сигнализировать об окончании сессии; тогда серверу пришлось бы отслеживать это значение в своем главном цикле.

Порядок, в котором мы создаем потоки для передачи объектов, тоже имеет значение. Сначала нужно получить исходящий поток, так как с помощью его заголовка конструктор объекта `ObjectInputStream` проверяет, действительно ли экземпляр `InputStream` способен передавать объекты. Если бы входящий поток был создан первым, обе стороны пытались бы получить заголовок друг у друга, что привело бы к взаимному блокированию. Теперь запустим наш сервер, указав в качестве параметра номер порта:

```
% java Server 1234
```

Чтобы запустить клиентское приложение, укажите имя сервера и тот же порт:

```
% java Client flatland 1234
```

Результат должен выглядеть примерно так:

```
Sun Oct 20 14:25:25 EEST 2013
4
```

Хорошо, может результат и не впечатляет, зато теперь нам есть от чего отталкиваться при разработке более реальных приложений. Представьте, что вам необходимо обрабатывать большие объемы данных. Сериализация позволяет обращаться с данными, как с обычными объектами, делая их передачу по сети тривиальной. Вам больше не нужно иметь дело с протоколами, которые работают на уровне байтов.

Ограничения

Как уже упоминалось ранее, во всей этой ситуации есть один подвох: клиент и сервер должны иметь доступ ко всем классам, которые им необходимы. То есть все классы запросов, включая `MyCalculation` (который

на самом деле является свойством класса `Client`), должны храниться в директории classpath как на клиентском, так и на серверном компьютере. В следующем разделе вы увидите, что вместе с сериализованными данными можно передавать байт-код классов, что позволяет динамически обмениваться по сети совершенно новыми видами объектов. Мы могли бы самостоятельно реализовать такой подход, воспользовавшись сетевым загрузчиком классов. Но делать это не обязательно: для этих целей платформа Java предлагает готовую технологию под названием RMI. Возможность передачи по сети как сериализованной информации, так и определений классов, востребована далеко не всегда, но благодаря ей платформа Java является мощным инструментом для прототипирования и разработки сложных распределенных приложений.

Удаленный вызов методов

Вызов методов — это наиболее фундаментальный механизм коммуникации на платформе Java. На его основе, к примеру, построена событийная модель, суть которой заключается в вызове методов из объектов, расположенных в одной виртуальной машине. Таким образом, если возникнет необходимость наладить связь между двумя Java-машинами на разных компьютерах, будет вполне естественно воспользоваться средствами с похожими возможностями, которые работают по принципу «запустить метод оттуда-то». Именно этим и занимается технология RMI (Remote Method Invocation — удаленный вызов методов). Она позволяет получить ссылку на объект, который находится на удаленном компьютере, и использовать ее так, как будто все происходит в рамках нашей локальной виртуальной машины. Благодаря этому мы можем выполнять методы на удаленном сервере, передавая им реальные значения в качестве аргументов, и получать в результате настоящие локальные объекты.

В этом подходе нет ничего нового. В языке C удаленный вызов процедур (Remote Procedure Calls или RPC) используется на протяжении многих лет; с его помощью программисты выполняют функции на удаленном компьютере, получая результат обратно. Однако в отличие от RMI, технология RPC в основном сосредоточена на работе со структурами данных. На платформе Java мы занимаемся не просто упаковкой и передачей информации — это относительно легкая задача; мы имеем дело с объектами, которые содержат как сами данные, так и методы для их обработки. Нам недостаточно просто передавать состояние объекта (данные) по сети, нам нужно, чтобы получатель смог взаимодействовать с этим объектом

(вызывать его методы). Технология RMI позволяет работать с сетевыми сервисами в объектном стиле, используя реальные, расширяемые типы, и передавая настоящие, «живые» ссылки между клиентом и сервером.

Нет никакого секрета в том, что технология RMI использует сериализацию; благодаря этому мы можем передавать полноценную иерархию объектов (не только сам объект, но и все другие объекты, на которые он ссылается). При необходимости RMI может применять динамическую загрузку классов, а также использовать диспетчер безопасности для надежной передачи байт-кода. Помимо того, что вызывать удаленные методы становится так же легко, как и локальные, мы получаем возможность передавать по сети не только данные, но и поведение класса (код).

Реальное применение

Теперь, воодушевившись знакомством с новой технологией, мы можем применить наши знания на практике. И хотя вся эта идея выглядит довольно многообещающей, обычные приложения редко пользуются возможностями RMI. Главными средствами передачи данных по сети уже много лет являются веб-сервисы, основанные на таких инструментах, как RPC, XML и HTTP. Благодаря своей кроссплатформенности эти стандарты могут легко адаптироваться для использования внутри веб-браузеров посредством языка JavaScript — в этом залог их популярности. Кроме того, веб-сервисы, которые работают по протоколу HTTP, обычно не имеют проблем с обходом брандмауэров, так как они используют тот же механизм, что и любые веб-страницы. Инструменты для разработки приложений, в основе которых лежат веб-сервисы, давно стали зрелыми и простыми в использовании, поэтому программисты обычно выбирают их даже тогда, когда весь код пишется на языке Java (и когда технология RMI давала бы больше возможностей). В этом разделе мы покажем вам, на что способна технология RMI; тем не менее вам определенно стоит ознакомиться со следующими главами, посвященными веб-сервисам и веб-приложениям.

Удаленные и локальные объекты

Объект, который будет использоваться через интерфейс RMI, должен быть доступен для сериализации. Но это еще не все. Технология RMI делает удаленные объекты по-настоящему распределенными. Это означает, что удаленный объект может находиться как на локальном, так и на каком-то другом компьютере. Термин «удаленный» говорит о том,

что объект используется через специальный интерфейс, который можно передавать по сети. Удаленный объект, как и любой другой, передается с помощью ссылки. Вне зависимости от того, где эта ссылка используется, все методы вызываются из исходного объекта, который по-прежнему находится на своем компьютере. Если у вас есть ссылка на удаленный объект и вы запустите с ее помощью какой-либо метод, то выполнение кода будет происходить на удаленном компьютере.

С локальными объектами все проще: вы можете их сериализовать и переправить по сети, как мы это делали в предыдущем разделе. Но здесь есть один подвох: когда локальный объект передается по сети, он просто копируется, поэтому ссылка на него будет работать только в рамках одного компьютера. Локальные объекты передаются по значению, а не по ссылке. Такое поведение уместно для объектов, которые используются для хранения данных при транспортировке — например для клиентских запросов или серверных ответов из нашего предыдущего примера. Их еще называют объектами-значениями или *объектами для передачи данных* (Data Transfer `Object` или DTO).

Удаленные интерфейсы

Удаленные объекты реализуют специальный *удаленный интерфейс*, который определяет, какие методы объекта можно вызывать по сети. Речь идет об интерфейсе `java.rmi.Remote`, который наследуется при создании приложения. Реализация этого интерфейса выполняется самым обычным образом. В коде своего клиентского приложения мы будем ссылаться на удаленный объект именно как на интерфейс, а не как на экземпляр класса-реализации. Для наших задач (вызов методов) настоящий объект ничем не отличается от заглушки, которую получает клиент, так как все это реализации одного и того же интерфейса; на локальном уровне нас не должно волновать, на что именно указывает наша ссылка. Благодаря этому мы можем применять к удаленным объектам обычные возможности языка, такие как приведение типов. Конечно, публичные поля (переменные) удаленного объекта недоступны через интерфейс, поэтому, если вы хотите с ними работать, вам придется предусмотреть соответствующие методы.

Существует еще одно требование, которое отличает удаленные объекты от локальных. Все методы в удаленном интерфейсе должны объявлять о возможности возникновения исключения `java.rmi.RemoteException`. Это исключение (или одна из его производных) генерируется при возникновении любой ошибки, связанной с сетью (это может быть сбой сервера, разрыв сети или истечение времени ожидания).

Некоторые разработчики всячески пытаются обойти это ограничение. Однако оно было введено не просто так — поведение удаленных и локальных объектов может отличаться, и ваш код должен это учитывать. Не существует «серебряной пули» (автоматические повторения операций, транзакции), которая бы полностью нивелировала все различия.

Ниже представлен простой пример удаленного интерфейса, который определяет поведение объекта `RemoteObject`. Мы создали два метода, которые можно вызывать удаленно; оба они возвращают некое значение типа `Value`.

```
import java.rmi.*;
public interface RemoteObject extends Remote {
   public Value doSomething() throws RemoteException;
   public Value doSomethingElse() throws RemoteException;
}
```

Экспорт удаленных объектов

Чтобы сделать удаленный объект доступным для других узлов в сети, нужно использовать класс `java.rmi.server.UnicastRemoteObject`. Вы можете сделать так, чтобы реализация вашего удаленного объекта была его потомком. Когда технология RMI начинает отслеживать подключения со стороны клиентов, она автоматически «экспортирует» производные этого класса. По аналогии с `java.lang.Object` этот базовый класс предоставляет реализации методов `equals()`, `hashcode()` и `toString()`, которые могут пригодиться удаленному объекту.

Далее показан пример удаленного объекта, который не только реализует интерфейс `RemoteObject`, рассмотренный нами ранее, но и наследует класс `UnicastRemoteObject`; мы намеренно опускаем реализацию двух методов и конструктора:

```
public class MyRemoteObject implements RemoteObject
      extends java.rmi.UnicastRemoteObject
{
   public MyRemoteObject() throws RemoteException {...}
   public Value doSomething() throws RemoteException {...}
   public Value doSomethingElse() throws RemoteException {...}
   // локальные методы
   private void doSomethingInternal() { ... }
}
```

Обратите внимание: мы должны предоставить конструктор (любой — даже пустой), который может генерировать исключение `RemoteException`, потому что так делает стандартный конструктор класса `UnicastRemoteObject`. Дело в том, что язык Java всегда обращается к конструктору родительского класса, даже если этого не видно. Наш класс может иметь любое количество дополнительных локальных методов (вероятно, большинство из них будут частными, хотя это не является строгим требованием), и для них не обязательно указывать исключение `RemoteException`.

Но что делать, если мы не можем (или не хотим) наследовать нашу реализацию удаленного объекта от класса `UnicastRemoteObject`? Допустим, нам нужно, чтобы это была производная класса `BankAccount` или какого-то специального базового типа в нашей системе. В таком случае мы можем сами заняться экспортом объекта, воспользовавшись статическим методом `exportObject()` из класса `UnicastRemoteObject`. Этот метод принимает в качестве аргумента интерфейс `Remote` и выполняет то, что обычно делает за нас конструктор `UnicastRemoteObject`. Закончив работу, он возвращает клиентскую заглушку для удаленного объекта. Но на практике вам вряд ли придется делать все это вручную. В следующем разделе вы узнаете, что на самом деле клиенты находят сервис при помощи реестра удаленных объектов (поисковый сервис, предоставляемый технологией RMI).

По умолчанию экспортированные объекты ждут подключения на *временном* (случайно выбранном) порту, хотя это зависит от конкретной реализации. Вы можете экспортировать объект с помощью другой версии метода `UnicastRemoteObject.exportObject()`, которая помимо экземпляра `Remote` принимает явно заданный номер порта.

И, наконец, само название «`UnicastRemoteObject`» может вызвать у вас вопрос: а какие еще бывают виды удаленных объектов? Сейчас их всего несколько. Есть тип под названием `Activatable`, который требует постоянного доступа в течение долгого времени. Позже в этой главе мы еще затронем тему активации интерфейсов RMI, но в подробности вникать не будем.

Реестр удаленных объектов

Реестр в технологии RMI играет роль своеобразного телефонного справочника. С его помощью можно искать удаленные объекты, зарегистрированные на другом компьютере, указывая имя, которое зависит

от конкретного приложения. Мы уже рассказывали, как с помощью удаленных методов можно передавать ссылки в разных направлениях. Реестр помогает автоматизировать этот процесс, позволяя клиенту искать исходный объект на удаленном компьютере.

Реестр реализован в виде класса `Naming` и утилиты под названием *rmiregistry*. Если ваше приложение хочет предоставить доступ к своим объектам через реестр, эта утилита должна быть предварительно запущена на том же компьютере. Если все сделано правильно, вы сможете создавать экземпляры удаленных объектов и привязывать их к конкретным именам в реестре. Какое бы имя вы ни выбрали, оно примет вид пути, разделенного слешами. Когда клиент захочет найти ваш объект, он сформирует специальный URL-адрес, состоящий из протокола «rmi:», имени сервера и названия объекта. После этого класс `Naming` на клиентской стороне свяжется с реестром и вернет ссылку на удаленный объект.

Так какой же объект должен попасть в реестр? В принципе, это может быть любой объект, для которого это единственный способ стать доступным для клиента. Вызов удаленного метода может вернуть другой удаленный объект без участия реестра. Кроме того, такой вызов может принимать удаленный объект в качестве аргумента, и при этом реестр тоже не требуется. Вы можете спроектировать свою систему так, чтобы в реестр попадал только один объект, который в дальнейшем будет служить фабрикой для всех последующих удаленных компонентов. То есть вы легко можете создать фабрику, которая будет возвращать ссылки на все удаленные объекты вашего приложения. В программах с определенной структурой это может получиться само собой.

Реестр, предоставляемый технологией RMI — это всего лишь одна из реализаций механизма поиска удаленных объектов (к слову, не очень изящная и с довольно медленными запросами). Не стоит рассматривать его как универсальное средство для иерархического представления ресурсов; это всего лишь способ организации взаимодействия в рамках технологии RMI.

Существуют и другие популярные сервисы для именования, которые предоставляют возможности подобного рода, и доступ к ним выполняется с помощью API-интерфейса JNDI (Java Naming and Directory Interface — сервис каталогов для организации и просмотра объектов по их именам). Интерфейс JNDI используется совместно с технологией RMI как часть спецификации Enterprise JavaBeans.

Пример использования технологии RMI

В первом примере использования технологии RMI мы просто продублируем протокол передачи сериализованных объектов из предыдущего раздела. Мы создадим удаленный объект под названием `MyServer`, с помощью которого можно будет вызывать методы, получать объекты `Date` и выполнять запросы `WorkRequest`. Сначала определим нашу версию интерфейса `Remote`:

```
//файл: ServerRemote.java
import java.rmi.*;
import java.util.*;
public interface ServerRemote extends Remote {
   Date getDate() throws RemoteException;
   Object execute( WorkRequest work ) throws RemoteException;
}
```

Интерфейс `ServerRemote` наследует `java.rmi.Remote`; объекты, которые его реализуют, будут считаться удаленными. Мы указали два метода, которые имели место в нашем старом протоколе: `getDate()` и `execute()`.

Далее мы создадим класс `MyServer`, который реализует данный интерфейс вместе с вышеупомянутыми методами. (К названиям реализаций удаленных интерфейсов часто добавляют суффикс «Impl». Если бы мы соблюдали это соглашение об именовании, наш класс мог бы называться `ServerImpl`.)

```
//файл: MyServer.java
import java.rmi.*;
import java.util.*;
public class MyServer
extends java.rmi.server.UnicastRemoteObject
implements ServerRemote {
   public MyServer() throws RemoteException { }
   // реализуем интерфейс ServerRemote
   public Date getDate() throws RemoteException {
      return new Date();
   }
   public Object execute( WorkRequest work )
         throws RemoteException {
      return work.execute();
   }
```

```
    public static void main(String args[]) {
        try {
            ServerRemote server = new MyServer();
            Naming.rebind("NiftyServer", server);
        } catch (java.io.IOException e) {
            // проблема при регистрации сервера
        }
    }
}
```

MyServer является наследником класса UnicastRemoteObject, поэтому любой его экземпляр автоматически экспортируется и начинает отслеживать сетевые подключения. Сначала мы описали конструктор, который должен генерировать исключение RemoteException, отвечающее за потенциальные ошибки во время экспорта объекта. Затем мы предоставили довольно простые реализации методов удаленного интерфейса ServerRemote.

Класс заканчивается методом main(), с помощью которого объект подготавливается к работе; в нем мы создаем экземпляр класса MyServer, и затем помещаем его в реестр, используя статический метод Naming.rebind(). Метод rebind() принимает два аргумента: имя, под которым зарегистрирован удаленный объект (NiftyServer — именно его использует клиент), и ссылка на экземпляр MyServer. Мы могли бы воспользоваться методом bind(), но он не рассчитан на ситуации, когда в реестре уже содержится объект с указанным именем (rebind() просто вставляет новый объект на место старого).

Если бы клиенты не использовали реестр для поиска сервера, нам бы не понадобилось создавать метод main() и выполнять все эти манипуляции с именами — доступ к удаленному объекту можно получить и без этого. Тогда единственным средством взаимодействия были бы аргументы и результаты выполнения методов — но, как мы уже упоминали ранее, это было бы частью другого шаблона проектирования под названием «фабрика».

Теперь создадим клиентскую часть:

```
//файл: MyClient.java
import java.rmi.*;
import java.util.*;
public class MyClient {
   public static void main(String [] args)
```

```
        throws RemoteException {
    new MyClient( args[0] );
}
public MyClient(String host) {
    try {
        ServerRemote server = (ServerRemote)
            Naming.lookup("rmi://"+host+"/NiftyServer");
        System.out.println( server.getDate() );
        System.out.println(
            server.execute( new MyCalculation(2) ) );
    } catch (java.io.IOException e) {
        // Ошибка ввода/вывода или некорректный URL-адрес
        } catch (NotBoundException e) {
            // объект NiftyServer не зарегистрирован
        }
    }
}
```

При запуске приложения MyClient нужно указать имя сервера, на котором работает реестр. Метод `main()` создает экземпляр класса `MyClient` и передает в его конструктор имя сервера, взятое из командной строки.

Имя сервера используется конструктором для создания URL-адреса объекта, который должен выглядеть примерно так: **rmi://hostname/NiftyServer**. (Не забывайте, `NiftyServer` — это имя, под которым зарегистрирован объект `ServerRemote`.) Позже этот адрес передается в статический метод `Naming.lookup()`. Если все пройдет хорошо, мы получим ссылку на объект `ServerRemote` (удаленный интерфейс). Реестр не имеет никакого понятия о том, какой именно объект он возвращает; таким образом, из метода `lookup()` мы получаем экземпляр `Object`, который нужно самостоятельно привести к типу удаленного интерфейса `ServerRemote`.

Запускаем приложение

Вы можете запускать клиент и сервер как локально, так и на разных компьютерах. Прежде всего, убедитесь в том, что все ваши классы находятся в директории classpath (или, если таковой не имеется, в текущей директории), после чего запустите на серверной стороне утилиту *rmiregistry* и приложение MyServer:

```
% rmiregistry & (в Windows: start rmiregistry)
%java MyServer
```

Затем запустите клиент, передав ему имя сервера (или «localhost», если все происходит на локальном компьютере):

```
% java MyClient myhost
```

Клиент должен вывести на экран дату и число 4, любезно вычисленные сервером. Ура! Вам удалось создать серьезное клиент-серверное приложение с помощью всего нескольких строчек кода.

Динамическая загрузка классов

Мы уже говорили, что перед запуском приложения следует скопировать все class-файлы как на клиентский, так и на серверный компьютер. Однако этого можно не делать, ведь технология RMI подходит для передачи по сети не только данных, но и классов. Давайте сделаем так, чтобы наши классы загружались по мере необходимости. Для этого нужно выполнить несколько дополнительных шагов.

Прежде всего, нам нужно сообщить технологии RMI, где искать недостающие классы. При запуске клиента или сервера можно указать URL-адрес хранилища (HTTP- или FTP-сервера), воспользовавшись системным свойством `java.rmi.server.codebase`. Это должен быть адрес JAR-архива или директории, где будет происходить поиск классов. Он будет передаваться вместе с сериализованным объектом (то есть, с его состоянием). Если помимо данных получателю понадобится class-файл, он получит его именно по этому адресу. Для динамической загрузки доступны любые классы, которые упоминаются в удаленном объекте (не только класс-заглушка). Это означает, что нам больше не нужно передавать на клиентскую сторону множество class-файлов; если они понадобятся, клиент сам сможет их загрузить. На рис. 13.3 мы видим пример того, как приложение MyClient обращается в реестр за ссылкой на объект `ServerRemote`. При этом класс-заглушка для объекта `MyServer` автоматически загружается с веб-сервера, который работает на том же компьютере, что и удаленный объект.

Теперь мы можем более логично разделять наши class-файлы между клиентским и серверным компьютерами. К примеру, мы можем не копировать на сервер класс `MyCalculation`, так как он на самом деле принадлежит клиенту. Вместо этого мы можем сделать его доступным через веб-сервер (который, скорее всего, будет запущен на одном компьютере с клиентом) и при запуске приложения MyClient указать соответствующий URL-адрес:

```
% java -Djava.rmi.server.codebase='http://myserver/foo/'...
```

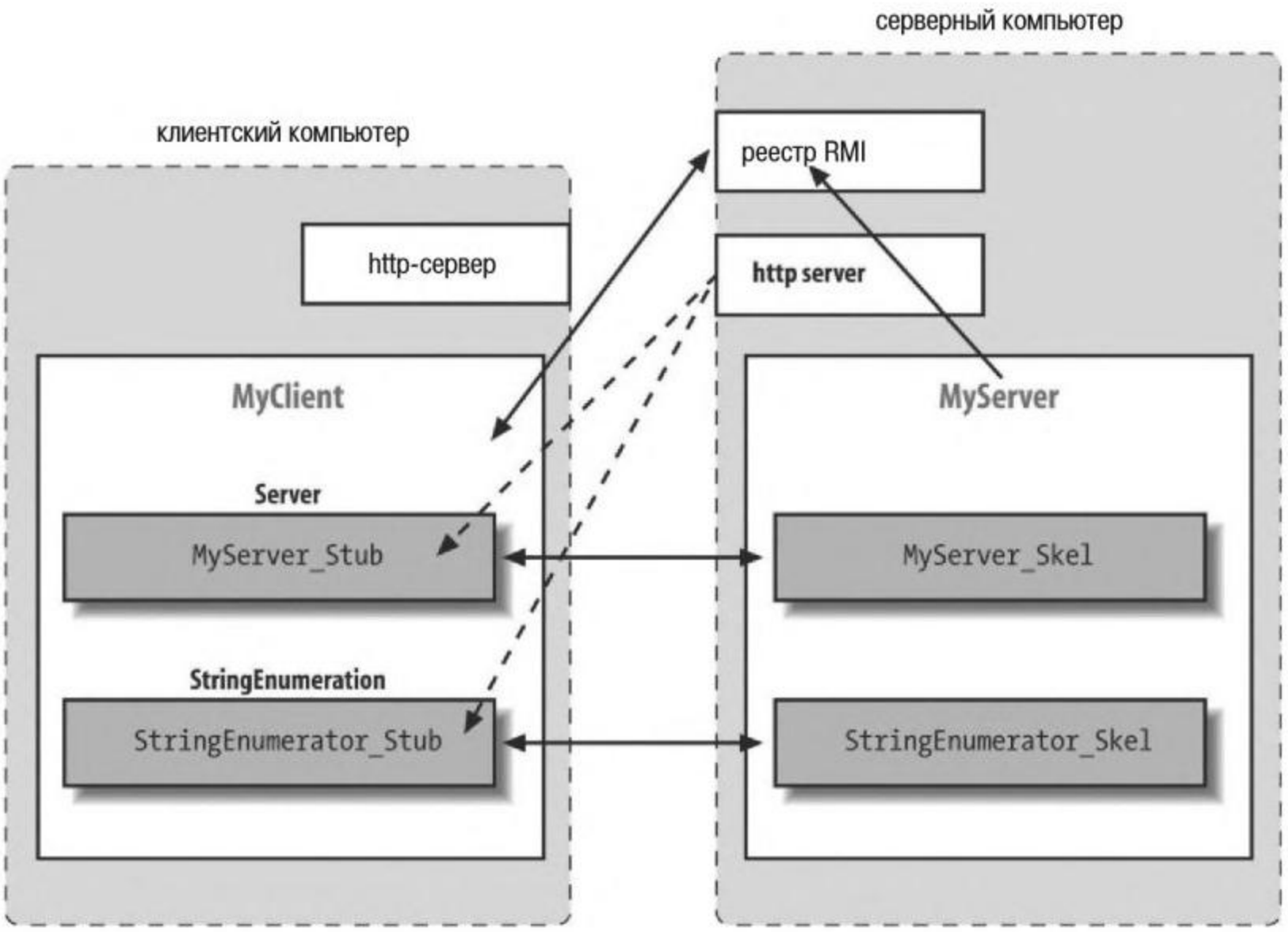

Рис. 13.3. Приложения на основе технологии RMI и динамическая загрузка классов

Замыкающий слеш в URL-адресе указывает на то, что путь ведет к директории, которая содержит class-файлы. В данном случае мы исходим из того, что класс `MyCalculation` доступен по адресу **http://myserver/foo/MyCalculation.class**.

Теперь пришло время позаботиться о безопасности. Мы вызываем методы из классов, загруженных по сети, поэтому нам нужно, чтобы диспетчер безопасности ограничивал диапазон действий, которые можно выполнять таким образом (по крайней мере, в тех случаях, когда код был получен не из надежного источника). При отсутствии диспетчера безопасности технология RMI отключает динамическую загрузку классов. Чтобы решить эту проблему, можно воспользоваться объектом `RMISecurityManager`. Это одна из разновидностей `SecurityManager`, которая применяет стандартную политику безопасности и вводит некоторые элементарные ограничения для загружаемых классов. Чтобы клиентские и серверные приложения могли использовать этот объект, достаточно добавить в начале их главных методов следующую строчку (как вы уже могли догадаться, в следующем разделе мы будем передавать код в обоих направлениях):

```
main() {
   System.setSecurityManager( new RMISecurityManager() );
   ...
```

Объект `RMISecurityManager` вводит ограничения согласно политике безопасности, описанной в соответствующем файле. Вы должны сами предоставить этот файл, чтобы ваши клиент и сервер могли выполнять базовые операции, такие как соединение по сети. К сожалению, чтобы разрешить динамическую загрузку классов, необходимо указать множество полномочий — мы не хотим сейчас в это углубляться. Будем считать, что вы не против предоставить этому коду полную свободу действий. Вот пример файла с описанием политики безопасности; назовем его **mysecurity.policy**:

```
grant {
      permission java.security.AllPermission ;
};
```

(Устанавливать диспетчер безопасности и затем снимать все ограничения — это крайне нелогично, если не сказать рискованно, но сейчас нас больше интересует код для работы с сетью.)

Для запуска приложения MyServer потребуется команда следующего вида:

```
% java -Djava.rmi.server.codebase='http://myserver/foo/' \
   -Djava.security.policy=mysecurity.policy MyServer
```

Чтобы разрешить динамическую загрузку классов, нам осталось сделать еще кое-что. В текущей реализации утилита *rmiregistry* не должна иметь возможность загружать class-файлы из директории classpath, иначе она просто не будет передавать вместе с сериализованными объектами URL-адреса их классов (и тогда никакой динамической загрузки не произойдет). Нам остается только предупредить вас об этом довольно досадном моменте.

Если вы выполнили все инструкции, для запуска клиента вам достаточно будет скопировать в директорию classpath класс `MyClient` и удаленный интерфейс `ServerRemote`. Все остальные классы по мере необходимости будут динамически загружаться с указанного сервера.

Передача ссылок на удаленные объекты

Пока что мы не показали вам ничего такого, чего нельзя было бы сделать с помощью простого протокола обмена объектами. Мы использовали всего один удаленный объект, ссылка на который была получе-

на из реестра RMI. Давайте расширим наш пример: организуем обмен удаленными ссылками между клиентом и сервером, позволив делать дополнительные уделанные вызовы в обоих направлениях. Добавим в интерфейс ServerRemote два новых метода:

```
public interface ServerRemote extends Remote {
   ...
   StringIterator getList() throws RemoteException;
   void asyncExecute( WorkRequest work, WorkListener
   listener )
      throws RemoteException;
}
```

Метод getList() извлекает из сервера объект нового для нас типа — StringIterator. Это простой список строк с некоторыми методами для последовательного доступа к элементам. Мы сделали его удаленным, чтобы он оставался на сервере.

С помощью метода asyncExecute() мы усовершенствовали наш подход к выполнению удаленных задач. Как и прежде, мы передаем запрос WorkRequest, но теперь вычисления будут проводиться независимо от клиента. Метод asyncExecute() имеет тип void, так как он ничего не возвращает; результат его выполнения мы получим позже. Вместе с запросом наш клиент передает ссылку на объект WorkListener, который даст нам знать, когда запрос WorkRequest завершит свою работу. В качестве реализации WorkListener будет выступать сам клиент.

Так как этот объект будет удаленным, он должен наследовать интерфейс Remote, а его методы обязаны предусматривать возможность возникновения исключений типа RemoteException:

```
//файл: StringIterator.java
import java.rmi.*;
public interface StringIterator extends Remote {
   public boolean hasNext() throws RemoteException;
   public String next() throws RemoteException;
}
```

Далее мы предоставляем простую реализацию интерфейса StringIterator под названием MyStringIterator:

```
//файл: MyStringIterator.java
import java.rmi.*;
```

```
public class MyStringIterator
      extends java.rmi.server.UnicastRemoteObject
      implements StringIterator {
   String [] list;
   int index = 0;
   public MyStringIterator( String [] list )
         throws RemoteException {
      this.list = list;
   }
   public boolean hasNext() throws RemoteException {
      return index < list.length;
   }
   public String next() throws RemoteException {
      return list[index++];
   }
}
```

`MyStringIterator` является наследником класса `UnicastRemoteObject`. Те простые методы, которые он содержит, позволяют получить следующую строку в списке и узнать, остались ли еще элементы для перебора.

Теперь давайте обсудим удаленный интерфейс `WorkListener`, который определяет то, как именно нужно отслеживать завершение работы объекта `WorkRequest`. Он содержит всего один метод, `workCompleted()`, который вызывается удаленным объектом сразу после выполнения задачи:

```
//файл: WorkListener.java
import java.rmi.*;
public interface WorkListener extends Remote {
   public void workCompleted(WorkRequest request, Object result )
      throws RemoteException;
}
```

Теперь давайте расширим возможности класса `MyServer`. Нам нужно реализовать методы `getList()` и `asyncExecute()`, которые мы только что определили в интерфейсе `ServerRemote`:

```
public class MyServer extends java.rmi.server.UnicastRemoteObject
            implements ServerRemote {
   ...
   public StringIterator getList() throws RemoteException {
      return new MyStringIterator(
         new String [] { "Foo", "Bar", "Gee" } );
```

```
}
public void asyncExecute(
   final WorkRequest request, final WorkListener listener )
   throws java.rmi.RemoteException
{
   new Thread() {
      public void run() {
         Object result = request.execute();
         try {
            listener.workCompleted( request, result );
         } catch ( RemoteException e ) {
            // ошибка при вызове клиента
            System.out.println( e );
         }
      }}.start();
   }
}
```

Метод `getList()` возвращает объект `StringIterator` с какими-то данными внутри. Метод `asyncExecute()` делает вызов `execute()` из объекта `WorkRequest` и оповещает реализацию `WorkListener` о завершении работы. Метод `asyncExecute()` выполняет запрос в отдельном потоке, благодаря чему удаленный вызов отрабатывает мгновенно. Позже, когда работа будет сделана, сервер вернет результат с помощью объекта `WorkListener`, размещенного на клиентской стороне.

Мы должны отредактировать класс `MyClient`, чтобы он реализовывал удаленный интерфейс `WorkListener`. Наш клиент превратится в удаленный объект, поэтому он, кроме всего прочего, должен наследовать класс `UnicastRemoteObject`. Нам также нужно добавить метод `workCompleted()`, который входит в состав интерфейса `WorkListener`. Чтобы опробовать на практике возможности нашего нового клиента, создадим для него отдельный класс под названием `MyClientAsync`:

```
//файл: MyClientAsync.java
import java.rmi.*;
import java.util.*;
public class MyClientAsync
   extends java.rmi.server.UnicastRemoteObject implements WorkListener
{
   public MyClientAsync(String host) throws RemoteException
   {
```

```
        try {
            RemoteServer server = (RemoteServer)
                Naming.lookup("rmi://"+host+"/NiftyServer");
            StringIterator si = server.getList();
            while( si.hasNext() ) {
                System.out.println( si.next() );
            }
            server.asyncExecute( new MyCalculation(100), this );
        } catch (java.io.IOException e) {
            System.out.println(e);
            // Ошибка ввода/вывода или некорректный URL-адрес
        } catch (NotBoundException e) {
            System.out.println(e);
            // Сервер NiftyServer не зарегистрирован
        }
    }
    public void workCompleted( WorkRequest request, Object result )
        throws RemoteException
    {
        System.out.println("Асинхронный результат: "+result );
    }
    public static void main(String [] args) throws RemoteException {
        new MyClientAsync( args[0] );
    }
}
```

С помощью метода `getList()` мы запрашиваем у сервера итератор, и затем выводим в цикле все строки. Чтобы выполнить математические вычисления, вызывается метод `asyncExecute()`; на этот раз мы возводим в квадрат число 100. В качестве второго аргумента для метода `asyncExecute()` выступает объект `WorkListener`, который сообщает о готовности данных; здесь наш клиент передает ссылку на самого себя **(this)**.

Перезагрузите на сервере реестр удаленных объектов и приложение MyServer, а также запустите где-нибудь свой клиент. Вы должны получить следующий результат:

```
Foo
Bar
Gee
Асинхронный результат = 10000
```

Надеемся, благодаря этому примеру вы смогли ощутить всю мощь технологии RMI, основанную на сериализации объектов и динамической загрузке классов. Язык Java одним из первых начал предлагать для создания распределенных приложений инструмент такого масштаба. И хотя некоторые из продвинутых возможностей RMI не нашли применения в прикладных программах, эта технология до сих пор имеет большое значение, являясь основой для широко распространенной архитектуры J2EE Enterprise JavaBeans. Больше информации о технологиях RMI и J2EE можно найти в книге «Java Enterprise in a Nutshell».

RMI и CORBA

На платформе Java у технологии RMI есть полноценная альтернатива под названием CORBA (Common `Object` Request Broker Architecture — общая архитектура брокера объектных запросов). Мы не будем ее здесь подробно рассматривать, но вы должны хотя бы знать о ее существовании. Это старый стандарт распределения объектов, разработанный консорциумом `Object` Management Group (OMG), одним из основателей которого является компания Sun Microsystems. Главным преимуществом технологии CORBA является ее «многоязычность»: с ее помощью Java-приложение может общаться с объектами, написанными на других языках, таких как C или C++. Это может оказаться существенным преимуществом в случаях, когда программа создается на основе уже существующей (и более старой) системы, переписывать которую будет слишком затратно. Технология CORBA частично дублирует сервисы, которые платформа Java Enterprise предоставляет через свои API-интерфейсы. Главным недостатком технологии CORBA является ее сложная, лишенная изящества и даже, в некотором роде, загадочная архитектура.

Масштабируемый ввод/вывод на основе пакета NIO

В данном разделе мы завершим обсуждение пакета NIO, начатое еще в главе 12. На этот раз мы сосредоточимся на неблокирующем и выборочном сетевом взаимодействии. Все примеры серверных приложений в этой главе использовали привязку к потокам выполнения (одна операция ввода/вывода на поток). Благодаря легкости, с которой можно создавать потоки на платформе Java, этот подход является вполне естественным. Он также очень эффективен, до определенной степени. Однако при построении крупномасштабных серверов такая

модель работы с клиентами порождает проблемы. Конечно, мощный сервер способен справиться с сотнями и даже тысячами потоков выполнения (особенно если большую часть времени они простаивают в ожидании ввода/вывода), но с точки зрения ресурсов это крайне прожорливое решение. Каждый поток, который вы запускаете на платформе Java, потребляет память для нужд своего внутреннего стека, и производительность приложения с таким количеством потоков сильно зависит от системы.

В качестве альтернативы можно вспомнить старые, смутные времена, когда многопоточность еще не была доступна и для управления множеством соединений использовались неблокирующие операции ввода/вывода, выполняющиеся в одном потоке. Давайте воспользуемся преимуществами многопроцессорных компьютеров и объединим этот классический подход с настраиваемым пулом потоков.

В основе этого механизма лежит *выборочный* ввод/вывод. Одних неблокирующих операций недостаточно, если у вас нет возможности управлять порядком их выполнения. Пакет NIO предоставляет эффективный механизм управления на основе выборочных каналов. Эти каналы позволяют регистрировать *селекторы* (специальный вид объектов для отслеживания различных операций), которые могут проверять готовность канала к выполнению таких операций как чтение/запись или прием/создание сетевых соединений.

Селекторы и сам процесс выборки строго следуют всем соглашениям, принятым в языке C, что не совсем похоже на отслеживание событий, которым мы занимались раньше. В основном это сделано из соображений производительности; эти API-интерфейсы предназначены для работы в высоконагруженных серверах, поэтому они очень тесно привязаны к традиционным системным инструментам и мало заботятся об удобстве в использовании. Все вышесказанное в сочетании с некоторыми другими особенностями применения пакета NIO означает, что материала в этом разделе будет много, а серверное приложение, которое мы здесь создадим, станет одним из самых длинных и сложных примеров во всей книге. Не расстраивайтесь, если вам не удастся понять все сразу. Универсальные методики, представленные ранее в этой главе, подходят для написания большинства приложений; данный раздел вы можете отложить до тех пор, пока у вас не возникнет необходимость создавать сервисы для обработки очень большого количества одновременных клиентских запросов.

Выборочные каналы

Являясь реализацией интерфейса `SelectableChannel`, выборочный канал поддерживает неблокирующий режим и содержит API-интерфейс для максимально эффективного процесса выборки. Всего существует четыре реализации выборочных каналов; три из них предназначены для работы с сетью: `SocketChannel`, `ServerSocketChannel` и `DatagramChannel`. Четвертая реализация, `Pipe`, является конвейером (и может использоваться аналогичным образом для взаимодействия в рамках одной виртуальной машины).

Весь процесс основан на объекте `Selector`, который хранит информацию об определенном наборе выборочных каналов и позволяет определять их готовность к выполнению операций ввода/вывода (используя метод `select()`). Концептуально это выглядит просто: вы регистрируете один или несколько каналов и затем с помощью селектора узнаете, какие из них готовы к работе. На практике приходится учитывать несколько дополнительных моментов.

Прежде всего, селектор работает не с самими каналами, а с объектами `SelectionKey`, которые создаются автоматически при регистрации каналов. Они инкапсулируют как сам выборочный канал, так и сведения о типе операций (например, чтение или запись), которые мы собираемся отслеживать. Эта информация хранится в объекте `SelectionKey` в виде опций (которые еще называют «набор интересов») и при необходимости может быть изменена в любой момент. Объекты `SelectionKey` также используются для получения результатов выборки. Каждый вызов метода `select()` возвращает количество каналов, готовых к выполнению определенного вида операций ввода/вывода. Соответствующие ключи можно получить с помощью метода `selectedKeys()`. Каждый ключ содержит набор опций, которые указывают на то, какие именно из интересующих нас операций готовы к выполнению (их может быть несколько). Например, с помощью опций объекта `SelectionKey` можно задекларировать наше желание получить уведомление в момент, когда канал будет готов к чтению или записи. Получив такой ключ в результате выборки, мы можем быть уверены в том, что канал готов к выполнению как минимум одной из указанных нами операций; чтобы узнать, какие именно операции нам доступны, можно проверить набор опций ключа.

Прежде чем мы продолжим, нужно кое-что уточнить. Мы говорили, что каналы регистрируются для селекторов, но API-интерфейс устроен иначе. На самом деле это селектор регистрируется для одного или нескольких

каналов, которыми он потом будет управлять, что выглядит довольно запутанно. Поэтому лучше сделать вид, что все работает наоборот.

Использование селекторов

Объект `Selector` создается с помощью фабричного метода `Selector.open()`:

```
Selector selector = Selector.open();
```

Чтобы зарегистрировать каналы для селектора, необходимо перевести их в неблокирующий режим:

```
SelectableChannel channelA = // ...
channelA.configureBlocking( false );
```

Сама регистрация выглядит так:

```
int interestOps = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
SelectionKey key = channelA.register( selector, interestOps );
```

Зарегистрировав канал, мы получаем возможность указать для него интересующие нас операции (набор интересов). Это делается с помощью полей-констант из класса `SelectionKey`:

`OP_READ`

Готовность к чтению

`OP_WRITE`

Готовность к записи

`OP_CONNECT`

Готовность к подключению клиентского сокета

`OP_ACCEPT`

Готовность к подключению серверного сокета

Это битовые поля; вы можете применять к ним логическое ИЛИ, как в вышеприведенном примере, чтобы обозначить свою заинтересованность в нескольких видах операций.

Результатом работы метода `register()` является ключ `SelectionKey`, который в свою очередь содержит методы `interestOps()` и `cancel()`; пер-

вый позволяет в любой момент изменить опции, а второй — отменить регистрацию канала.

Тот же самый ключ возвращается и в результате выборки, если его канал готов. В этом случае объект `SelectionKey` будет содержать набор опций для операций, которые работают в неблокирующем режиме. Мы можем извлекать значения этих опций с помощью метода `readySet()`. Для проверки готовности всех видов операций предусмотрены отдельные вспомогательные методы: `isReadable()`, `isWritable()`, `isConnectable()` и `isAcceptable()`. Некоторым приложениям просто не нужно сохранять объект `SelectionKey` во время регистрации. В нашем примере селектор сам следит за ключами, а мы используем их только тогда, когда они уже готовы. Но мы на этом не остановились и сделали так, чтобы объект `SelectionKey` сам хранил ссылку! Его вспомогательный метод `attach()` позволяет прикрепить к ключу любой объект, который мы потом сможем использовать в нашем приложении. Чуть позже мы покажем, какую пользу из этого можно извлечь.

Зарегистрировав несколько каналов, мы можем сделать выборку с помощью метода `select()` из объекта `Selector`:

```
int readyCount = selector.select();
```

Если оставить этот метод без аргументов, он заблокирует поток выполнения и будет ждать, когда хотя бы один канал станет готовым для выполнения каких-либо операций, или пока из селектора не будет вызван метод `wakeup()`. Но вы можете указать время ожидания готовности канала (в миллисекундах), по истечении которого метод `select()` сам завершит свою работу. Есть еще один метод, `selectNow()`, который завершается мгновенно. В любом случае в качестве результата вы получите количество готовых каналов.

Операции `select()` и `wakeup()` можно использовать примерно так же, как методы `wait()` и `notify()`. Метод `wakeup()` необходим, ведь без него в процессе выборки нельзя будет поменять интересующие нас опции. Если какой-то другой поток изменит набор интересов, он должен будет вызвать метод `wakeup()`, чтобы выборка началась заново. В объекте `Selector` большое внимание уделяется синхронизации; например, регистрация новых каналов блокируется до тех пор, пока не отработает метод `select()`. Иногда намного проще указать время ожидания и запустить выборку внутри цикла:

```
while ( selector.select( 50 ) == 0 );
```

Но если другой поток может изменить опции, вам все равно придется вызывать метод `wakeup()`, чтобы максимально ускорить процесс. Иначе, в худшем случае, каждая итерация будет блокироваться на протяжении всего времени ожидания, даже если у селектора появится новая работа.

Сделав выборку, мы можем вызвать метод `selectedKeys()` и получить из селектора список готовых каналов, который можно перебирать и использовать на свое усмотрение:

```
Set readySet = selector.selectedKeys();
for( Iterator it = readySet.iterator(); it.hasNext(); ) {
   SelectionKey key = (SelectionKey)it.next();
   it.remove(); // удаляем ключ из набора
   // используем ключ
}
```

Готовые каналы возвращаются нам в виде набора `java.util.Set`, который можно перебрать с помощью итератора (см. главу 1). Обратите внимание: мы использовали метод `remove()` из объекта `Iterator`, чтобы убрать ключ из набора. Методы `select()` поддерживают исключительно добавление (либо ключа в набор готовых каналов, либо опцию в ключ из этого набора), поэтому при обработке ключей мы сами должны их убирать. Чтобы получить набор всех ключей, зарегистрированных для селектора, нужно вызвать метод `keys()`, но удалять ключи из этого набора не нужно; для отмены регистрации предусмотрен метод `cancel()`. Чтобы отменить регистрацию всех ключей сразу, можно закрыть сам селектор, используя метод `close()`.

Приложение LargerHttpd

Давайте применим все эти знания на практике. Помните TinyHttpd — веб-сервер, который мы написали в начале этой главы? В этом разделе у него появится старший брат по имени LargerHttpd. Это будет неблокирующий веб-сервер, который обслуживает запросы с помощью каналов `SocketChannel` и пула потоков выполнения. Главный цикл, работая в однопоточном режиме, будет принимать новые подключения и проверять, готовы ли к чтению или записи имеющиеся клиентские соединения. Каждый раз, когда клиенту что-то нужно, он размещает в очереди новую задачу, которая будет обслужена одним из потоков нашего пула. Как уже было сказано, данный пример получился немного длиннее, чем нам бы того хотелось, но это действительно минимальный объем кода,

с помощью которого можно продемонстрировать реальное использование изученных нами API-интерфейсов.

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.regex.*;
public class LargerHttpd
{
   Selector clientSelector;
   public void run( int port, int threads ) throws IOException
   {
      clientSelector = Selector.open();
      ServerSocketChannel ssc = ServerSocketChannel.open();
      ssc.configureBlocking(false);
      InetSocketAddress sa =   new InetSocketAddress( InetAddress
            .getLoopbackAddress(), port );
      ssc.socket().bind( sa );
      ssc.register( clientSelector, SelectionKey.OP_ACCEPT );

      Executor executor = Executors.newFixedThreadPool( threads );
      while ( true ) {
         try {
            while ( clientSelector.select(100) == 0 );
      Set<SelectionKey> readySet = clientSelector.selectedKeys();
   for(Iterator<SelectionKey> it=readySet.iterator(); it.hasNext();)
      {
         final SelectionKey key = it.next();
         it.remove();
         if ( key.isAcceptable() ) {
               acceptClient( ssc );
         } else {
               key.interestOps( 0 );
               executor.execute( new Runnable() {
                  public void run() {
                     try {
                           handleClient( key );
```

```
                    } catch ( IOException e) { System.out.println(e); }
                        }
                    } );
                }
            }
        } catch ( IOException e ) { System.out.println(e); }
    }
}
void acceptClient( ServerSocketChannel ssc ) throws IOException {
    SocketChannel clientSocket = ssc.accept();
    clientSocket.configureBlocking(false);
    SelectionKey key = clientSocket.register( clientSelector,
    SelectionKey.OP_READ );
    HttpdConnection client = new HttpdConnection( clientSocket );
    key.attach( client );
}
void handleClient( SelectionKey key ) throws IOException {
    HttpdConnection client = (HttpdConnection)key.attachment();
    if ( key.isReadable() ) {
        client.read( key );
            } else {
        client.write( key );
            }
    clientSelector.wakeup();
}
public static void main( String argv[] ) throws IOException {
    //new LargerHttpd().run( Integer.parseInt(argv[0]), 3/*threads*/ );
    new LargerHttpd().run( 1235, 3/*threads*/ );
    }
}
class HttpdConnection {
    static Charset charset = Charset.forName("8859_1");
    static Pattern httpGetPattern = Pattern.compile("(?s)GET /?(\\S*).*");
    SocketChannel clientSocket;
    ByteBuffer buff = ByteBuffer.allocateDirect( 64*1024 );
    String request;
    String response;
    FileChannel file;
    int filePosition;
    HttpdConnection ( SocketChannel clientSocket ) {
        this.clientSocket = clientSocket;
```

```
}
void read( SelectionKey key ) throws IOException {
   if ( request == null && (clientSocket.read( buff ) == -1
         || buff.get( buff.position()-1 ) == '\n' ) )
      processRequest( key );
   else
      key.interestOps( SelectionKey.OP_READ );
}
void processRequest( SelectionKey key ) {
   buff.flip();
   request = charset.decode( buff ).toString();
   Matcher get = httpGetPattern.matcher( request );
   if ( get.matches() ) {
      request = get.group(1);
      if ( request.endsWith("/") || request.equals("") )
         request = request + "index.html";
      System.out.println( "Запрос: "+request);
      try {
         file = new FileInputStream ( request ).getChannel();
      } catch ( FileNotFoundException e ) {
         response = "404 Объект не найден";
      }
   } else
      response = "400 Неверный запрос" ;
   if ( response != null ) {
      buff.clear();
      charset.newEncoder().encode(
         CharBuffer.wrap( response ), buff, true );
      buff.flip();
   }
   key.interestOps( SelectionKey.OP_WRITE );
}
void write( SelectionKey key ) throws IOException {
   if ( response != null ) {
      clientSocket.write( buff );
      if ( buff.remaining() == 0 )
         response = null;
   } else if ( file != null ) {
      int remaining = (int)file.size()-filePosition;
   long sent = file.transferTo( filePosition, remaining, clientSocket);
      if ( sent >= remaining || remaining <= 0 ) {
```

```
                file.close();
                file = null;
            } else
                filePosition += sent;
        }
        if ( response == null && file == null ) {
            clientSocket.close();
            key.cancel();
        } else
            key.interestOps( SelectionKey.OP_WRITE );
    }
}
```

На первый взгляд, приложение LargerHttpd имеет такую же структуру, как и TinyHttpd. Главный класс, `LargerHttpd`, принимает подключения, а класс `HttpdConnection` инкапсулирует сокет и отвечает за взаимодействие с клиентом. Но на этот раз мы не стали представлять соединение в виде объекта `Runnable` и помещать его в отдельный поток; вместо этого мы разделили его функции между двумя основными методами — `read()` и `write()`. Задача класса `LargerHttpd` заключается в приеме новых клиентских подключений, создании для них оберток типа `HttpdConnection` и отслеживании их состояния с помощью селектора. Обнаружив, что клиент готов к отправке или получению данных, мы передаем задачу `Runnable` для выполнения в объекте `Executor`. В зависимости от типа доступной операции, из клиентского соединения вызываются методы `read()` или `write()`.

Объект `HttpConnection` инкапсулирует состояние взаимодействия с клиентом. Ввиду своего достаточно высокоуровневого интерфейса он должен следить за тем, что сейчас делает клиентская сторона: ждет поступления новых данных, генерирует ответ или производит запись в исходящий поток. Объект `HttpdConnection` также управляет опциями своего ключа, чтобы иметь возможность эффективно планировать свою активность с учетом готовности сокета к чтению или записи. Для связи с клиентским соединением в объекте `SelectionKey` предусмотрены методы `attach()` и `attachment()`.

Метод `acceptClient()` из класса `LargerHttpd` выполняет сразу несколько задач. Он принимает сокет с новым подключением, настраивает и регистрирует его для селектора, указывая стандартный набор опций (только чтение). Затем он создает объект `HttpdConnection` на основе сокета и привязывает его к ключу, чтобы позже мы могли его как-то извлечь.

Главный цикл класса `LargerHttpd` выглядит довольно незамысловато. Сначала мы настраиваем канал `ServerSocketChannel`. Эта процедура похожа на подготовку объекта `ServerSocket`, только здесь нам сначала нужно создать экземпляр `InetSocketAddress`, чтобы сохранить локальный адрес и порт нашего сервера и передать эту информацию сокету с помощью метода `bind()` из канала `ServerSocketChannel`. Мы также переводим серверный сокет в неблокирующий режим и регистрируем его для нашего главного селектора, чтобы иметь возможность делать выборку клиентских соединений в том же цикле, в котором мы получаем набор опций ключа, отвечающих за чтение или запись.

В главном цикле выборки мы проверяем, готов ли ключ принять подключение, и при условии положительного ответа вызываем метод `acceptClient()`; в противном случае мы обнуляем опции ключа с помощью метода `interestOps()` и передаем ключ в метод `handleClient()`, используя задачу `Runnable`. Очень важно обнулить опции перед началом следующего цикла, иначе все будет зависеть от того, успеет ли пул потоков завершить все свои задачи до того, как мы узнаем о наличии еще одного соединения, готового к работе. Обнуляя и сбрасывая опции ключа при завершении операции мы можем быть уверены в том, что каждое конкретное соединение обрабатывается только одним потоком выполнения.

Если операция готова к выполнению, мы передаем в объект `Executor` задачу, которая вызывает метод `handleClient()` с ключом в качестве аргумента. С помощью ключа мы получаем соответствующее соединение и вызываем метод, который подходит для текущей операции (чтение или запись). После этого объект `HttpdConnection` сам выполняет всю работу. В приложении, где каждое соединение привязывается к потоку, вызов метода `read()` был бы эквивалентен одной итерации цикла чтения. Каждый раз мы пытаемся прочитать все доступные данные, проверяя, достигнут ли конец строки (символ «\n»). Получив всю строку, мы вызываем метод `processRequest()`, который превращает байтовый буфер в текст и вычленяет из запроса путь к файлу, как это делал сервер TinyHttpd. Для каждого ключа с незавершенной операцией `read()` мы сбрасываем опции в начальное состояние (`OP_READ`). Завершая чтение и обработку запроса, мы переключаемся в режим `OP_WRITE`, так как теперь мы готовы отправить ответ.

Метод `write()` различает отправку текстовых ответов (сообщений об ошибке) и файлов с помощью полей `response` и `file`. Благодаря методу `transferTo()` из объекта `FileChannel` содержимое файла передается

сразу в сетевой сокет, минуя память Java-машины (этот небольшой веб-сервер и в самом деле эффективный). Вот, собственно, и все. Завершив запись, мы закрываем клиентский сокет и отменяем регистрацию ключа; в итоге, при следующей выборке ключ будет убран из набора, хранящегося в селекторе (что приведет к сбросу связанного с ним соединения `HttpdConnection`).

Неблокирующие операции на клиентской стороне

На примере типичного серверного приложения было продемонстрировано, как с помощью канала `SocketChannel` можно добиться неблокирующего, выборочного ввода/вывода. Неблокирующие операции в основном востребованы на серверах, но это вовсе не означает, что вы не можете их использовать на клиентской стороне — например, в приложениях, которые работают в децентрализованных сетях и обслуживают множество соединений в обоих направлениях.

Для клиентских приложений существует еще один дополнительный инструмент — неблокирующая операция подключения. Процесс создания TCP-соединения между клиентской стороной и удаленным сервером подразумевает двухэтапное подтверждение. При этом поток выполнения обычно блокируется до тех пор, пока соединение не будет установлено. Однако пакет NIO предлагает альтернативный подход, который позволяет следить за состоянием инициированного подключения. В неблокирующем режиме метод `connect()` из объекта `SocketChannel` завершает работу мгновенно, после чего попытка соединения (чем бы она ни закончилась) выполняется в фоне. Позже, проверяя наличие опции `OP_CONNECT` с помощью селектора, можно будет узнать, готов ли сокет установить соединение. В случае готовности из канала `SocketChannel` вызывается метод `finishConnect()`, который либо сразу завершается, либо, если произошла ошибка, генерирует исключение `IOException`. Здесь мы не столько выполняем работу, сколько узнаем об успешности (или провале) попытки асинхронного подключения.

Глава 14
ВЕБ-ПРОГРАММИРОВАНИЕ

При упоминании приставки «веб-» мы, как правило, представляем себе веб-приложения и веб-сервисы. Можно также вспомнить о веб-браузерах и веб-серверах, которые выполняют эти приложения и перемещают данные по сети. Но здесь важно понимать, что развитию Интернета способствовали именно стандарты и протоколы, а не программы и инструменты как таковые. Даже на самых ранних этапах существования глобальной сети можно было передавать файлы из одной точки в другую, просматривать документы, не уступавшие по своим возможностям HTML-страницам, и т. д. Но в те времена не существовало унифицированной модели для идентификации, получения и отображения данных, не было и универсального способа, с помощью которого приложения могли бы взаимодействовать с этими данными по сети. С началом взрывного роста Интернета формат HTML стал доминировать, и большинство разработчиков в наши дни имеют о нем как минимум общее представление. В этой главе мы поговорим о его двоюродном брате, протоколе HTTP, который отвечает за связь между веб-клиентами и серверами; мы также обсудим идентификаторы URL, которые предоставляют стандартный способ адресации объектов в сети. Платформа Java предлагает очень простой API-интерфейс для работы с URL-адресами. В этой главе вы узнаете, как написать веб-клиент, способный взаимодействовать с серверами по протоколу HTTP с использованием запросов типа GET и POST. Не останутся без внимания и веб-сервисы — следующее звено в эволюционной цепочке. В главе 15 мы рассмотрим противоположную сторону сетевого взаимодействия — серверную, в качестве которой будут выступать веб-сервисы и сервлеты, работающие в рамках веб-сервера.

Единый указатель ресурсов (URL)

URL-адрес указывает на объект в Интернете. Это текстовая строка, которая служит для идентификации элемента. Она говорит о том, где его найти, как с ним связаться и каким образом его можно извлечь. URL-адрес поддерживает любые источники информации. Это могут быть как статиче-

ские данные (файл, локальная файловая система или веб-сайт, размещенный на веб- или FTP-сервере), так и динамические объекты, такие как RSS-ленты или записи в базе данных. Поддерживаются и более интерактивные ресурсы — сессии для обмена данными и адреса электронной почты.

Существует множество способов определить местонахождение элемента в сети; для передачи разных видов информации требуются соответствующие инструменты и протоколы. В результате, URL-адреса могут выглядеть совершенно по-разному. Самая распространенная разновидность этого формата состоит из четырех компонентов: сетевой узел (сервер), имя элемента, его местонахождение в рамках сервера и протокол, по которому с этим сервером нужно взаимодействовать:

```
протокол://имя-узла/путь/имя-элемента
```

Протокол (его еще называют «схемой») — это идентификатор вроде http или ftp; за ним следует *имя узла* в сети (или доменное имя); дальше указывается *путь*, по которому *элемент* можно идентифицировать на сервере. URL-адрес может содержать и дополнительную информацию — например номер порта для сетевого протокола, или так называемый «якорь», который ссылается на отдельный раздел внутри документа. Существуют и другие, узкоспециализированные виды URL, такие как адреса электронной почты (mailto) или идентификаторы компонентов баз данных; нарушая некоторые положения этого формата, они, тем не менее, соблюдают его главный принцип, согласно которому адрес должен состоять из протокола и уникального идентификатора (некоторые из таких адресов было бы логичнее называть URI-идентификаторами, но об этом мы поговорим позже).

Большинство URL-адресов формируются по принципу иерархии или пути, поэтому иногда нам приходится иметь дело с относительными и *базовыми адресами*. Базовый адрес используется как отправная точка, относительно которой указывается вся дополнительная информация о местоположении объекта. Например, базовый URL-адрес может ссылаться на директорию, которая находится на веб-сервере, а относительный — на имя файла в этой (или дочерней) директории.

Класс `URL`

Теперь давайте перейдем к более конкретным вещам, а именно к классу `URL`. Этот класс представляет URL-адрес и содержит API-интерфейс для доступа к веб-ресурсам, таким как HTML-страницы и серверные

приложения. Он поддерживает расширяемый набор протоколов и обработчиков содержимого для налаживания связи и, теоретически, может использоваться для преобразования данных. Благодаря классу `URL` создание сетевого соединения с сервером и получение необходимой информации занимает всего несколько строчек кода. Чтобы обеспечить поддержку новых видов серверов и форматов данных, вы можете создавать дополнительные разновидности этого класса, не внося изменения в основной код своих программ.

На платформе Java URL-адрес представлен экземпляром класса `java.net.URL`. Объект `URL` управляет информацией обо всех составляющих адреса, позволяя получать идентифицируемый им ресурс. Мы можем сформировать объект `URL` из отдельных частей:

```
try {
   URL aDoc =
      new URL( "http://foo.bar.com/documents/homepage.html" );
   URL sameDoc =
      new URL("http","foo.bar.com","documents/homepage.html");
} catch ( MalformedURLException e ) { ... }
```

Эти два объекта указывают на один и тот же сетевой ресурс — страницу *homepage.html* на сервере **foo.bar.com**. Но узнать, существует ли ресурс на самом деле, можно только при попытке получить к нему доступ. Изначально объект `URL` содержит только сведения о местоположении ресурса и о том, как к нему подключиться. На этом этапе он не устанавливает никаких соединений с сервером. Мы можем извлекать различные составляющие URL-адреса с помощью методов `getProtocol()`, `getHost()` и `getFile()`. Метод `sameFile()` позволяет проверить, указывают ли два адреса на один и тот же ресурс (не очень удачное название, потому что в качестве ресурсов могут выступать не только файлы). Он не дает стопроцентной гарантии, так как это не просто сравнение двух строк; среди прочих факторов учитывается возможность того, что один сервер может иметь несколько имен (хотя до сравнения содержимого самих ресурсов дело не доходит).

При создании объекта `URL` в первую очередь определяется протокол. Если протокол некорректен или у платформы Java нет для него соответствующего обработчика, конструктор сразу генерирует исключение `MalformedURLException`. Под обработчиком имеется в виду специальный класс, который реализует сетевой протокол для доступа к ресурсу по заданному URL-адресу. К примеру, для протокола «`http`» Java-машина

создаст экземпляр класса `HttpURLConnection` и попытается с его помощью прочитать документ на указанном веб-сервере.

Платформа Java 7 содержит классы для работы с такими протоколами как `http`, `https` (безопасная версия HTTP), `ftp`, `file` (для локальной файловой системы) и `jar` (для работы с файлами внутри JAR-архивов). Использование протоколов, не входящих в этот список, может оказаться довольно рискованным. Об этом и о проблемах, связанных с обработкой содержимого сетевых ресурсов, мы поговорим чуть позже.

Потоковая передача данных

Наиболее низкоуровневый и универсальный способ получения данных по URL-адресу связан с использованием метода `openStream()`, который позволяет получить из объекта `URL` поток `InputStream`. Передача данных в потоковом режиме может пригодиться в тех случаях, когда необходимо получать постоянные обновления из динамического источника. Недостаток этого подхода заключается в том, что вы сами должны разбирать содержимое потока байтов. Это практически то же самое, что работать с байтовыми потоками с помощью сокетов, но здесь все взаимодействие с сервером ложится на обработчик протокола, а вы будете иметь дело только с результатами транзакции. Метод `openStream()` предоставляется не для всех видов URL-адресов, так как не все они указывают на какой-то конкретный набор данных (в этом случае вы получите исключение `UnknownServiceException`).

Следующий код выводит содержимое HTML-страницы, хранящейся на веб-сервере:

```
try {
   URL url = new URL("http://server/index.html");
   BufferedReader bin = new BufferedReader (
      new InputStreamReader( url.openStream() ));
   String line;
   while ( (line = bin.readLine()) != null ) {
      System.out.println( line );
   }
   bin.close();
} catch (Exception e) { }
```

Поток `InputStream` полученный с помощью метода `openStream()`, помещается внутрь объекта `BufferedReader`, который поддерживает

построчное считывание текста. Так как в URL-адресе был указан протокол HTTP, для сетевого взаимодействия используется обработчик `HttpURLConnection`. Обратите внимание: мы пока еще даже не начинали обсуждать обработку самих данных, поскольку в этом примере чтение выполняется непосредственно из входящего потока (следовательно, информацию не нужно преобразовывать).

Получение данных в виде объекта

Как уже говорилось ранее, чтение байтов из потока — это наиболее универсальный способ доступа к данным по сети. Обработку полученных байтов метод `openStream()` оставляет за вами. Однако класс `URL` поддерживает более сложный, расширяемый механизм преобразования содержимого сетевых ресурсов, который мы обсудим прямо сейчас. Но нужно понимать, что ввиду отсутствия должной стандартизации и ограничений, связанных с созданием новых обработчиков, данный механизм не получил широкого распространения. И хотя за последние годы сообщество Java-программистов добилось некоторого прогресса в предоставлении стандартных обработчиков для небольшого набора протоколов, практическая ценность этого, безусловно, интересного раздела невелика.

Чтобы получить содержимое сетевого ресурса в виде обычного объекта, нужно знать, какой тип данных считывается по URL-адресу, и иметь для этого типа подходящий обработчик. Вот как это работает: метод `getContent()` инициирует соединение с удаленным узлом, автоматически считывает содержимое ресурса, определяет тип данных и вызывает соответствующий обработчик, который превращает байты в полноценный объект. Это похоже на процесс передачи сериализованного, который мы демонстрировали в главе 13. Чтобы определить тип полученных данных, платформа Java может проверить MIME-тип ресурса, расширение файла или даже содержимое потока байтов

Возьмем для примера адрес **http://foo.bar.com/index.html**. Для чтения данных метод `getContent()` воспользуется обработчиком `HttpURLConnection`, а чтобы превратить эти данные в подходящий объект, может понадобиться экземпляр класса `ContentHandler` для формата HTML. Точно так же изображение в формате GIF будет преобразовано в объект `ImageProducer` из пакета AWT. Если GIF-файл будет храниться на FTP-сервере, обработчик содержимого останется прежним — поменяется только обработчик протокола, который считывает данные.

Обработчик содержимого должен поддерживать любые объекты, поэтому метод `getContent()` возвращает универсальное значение типа `Object`. Чуть ниже мы покажем, как узнать MIME-тип ресурса, чтобы понять, какой же на самом деле объект мы получили. Имея в своем распоряжении все доступные нам сведения об объекте, мы можем привести его к подходящему (и более конкретному) типу. Например, если мы ожидаем получить изображение, результат выполнения метода `getContent()` можно привести к типу `ImageProducer`:

```
try {
   ImageProducer ip = (ImageProducer)myURL.getContent();
} catch ( ClassCastException e ) { ... }
```

При попытке получения данных может возникнуть целый ряд ошибок. Например, проблемы со связью приводят к тому, что метод `Content()` генерирует исключение `IOException`. Некоторые ошибки возникают на уровне приложения, поэтому необходимо знать, как разные виды объектов, которые работают с протоколами и содержимым, реагируют на нештатные ситуации. Для некоторых MIME-типов не предусмотрено соответствующего обработчика — в этом случае метод `getContent()` вызывает специальный обработчик для «неизвестного типа», который возвращает данные в виде обычного потока `InputStream` (в итоге мы приходим к тому, с чего начали).

Иногда нам могут понадобиться сведения об обработчике протокола. Представьте, что объект `URL` ссылается на файл, которого не существует. При запросе сервер вернет уже знакомое нам сообщение «404 Not Found». Чтобы как-то реагировать на подобные ситуации, связанные с конкретным протоколом, нужно обращаться непосредственно к обработчику (и ниже мы об этом поговорим).

Управление соединениями

При вызове методов `openStream()` или `getContent()` объект `URL` сначала обращается к обработчику протокола и только потом связывается с удаленным узлом. Сетевое соединение представлено классом `URLConnection`, производные которого отвечают за управление сетевым взаимодействием по заданному протоколу и предоставляют дополнительные метаданные, относящиеся к ресурсу. Например, класс `HttpURLConnection` управляет простыми веб-запросами, позволяя использовать некоторые особенности протокола HTTP, интерпретируя

сообщения «404 Not Found» и другие ошибки веб-сервера. Мы еще вернемся к этому классу в следующем разделе.

Экземпляр `URLConnection` можно получить непосредственно из объекта `URL`, воспользовавшись методом `openConnection()`. С его помощью мы можем сделать много интересных вещей — например, узнать MIME-тип ресурса, не считывая его содержимое:

```
URLConnection connection = myURL.openConnection();
String mimeType = connection.getContentType();
InputStream in = connection.getInputStream();
```

Несмотря на свое название, объект `URLConnection` изначально ни к чему не подключен. В этом примере сетевое соединение было установлено только после вызова метода `getContentType()`. Объект `URLConnection` не связывается с источником до тех пор, пока вы не запросите данные или не вызовите метод `connect()`. Перед этим вы имеете возможность уточнить различные сетевые параметры и функции, связанные с конкретным протоколом. К примеру, мы можем указать время ожидания как для подключения к серверу, так и для операций чтения:

```
URLConnection connection = myURL.openConnection();
connection.setConnectTimeout( 10000 ); // миллисекунды
connection.setReadTimeout( 10000 ); // миллисекунды
InputStream in = connection.getInputStream();
```

Объект `URLConnection` можно привести к определенному подтипу, чтобы получить информацию, связанную с интересующим нас протоколом. Об этом мы поговорим в разделе «Использование запросов типа POST».

Использование обработчиков в реальных задачах

Средства обработки протоколов и содержимого, которые мы описали выше, являются очень гибкими; для работы с новым типом URL-адреса вам достаточно предоставить соответствующий класс обработчика. Такой подход можно было бы применять в веб-браузере, чтобы загружать модули для поддержки отдельных протоколов прямо по сети. Эту идею пытались популяризировать на ранних этапах развития платформы Java — к сожалению, безуспешно. Для динамической загрузки обработчиков содержимого и протоколов необходимы специальные API-интерфейсы, а их до сих пор не существует. Более того, у нас даже нет стан-

дартных средств для определения того, какие обработчики доступны на заданной платформе.

На сегодняшний день существуют классы для работы с протоколами HTTP, HTTPS, FTP, FILE и JAR. И хотя они присутствуют практически во всех версиях платформы Java, стопроцентной гарантии никто не дает. С обработчиками содержимого все еще сложнее. К примеру, в стандартной библиотеке нет классов для обработки HTML, GIF, JPEG и других популярных форматов. Более того, некоторые обработчики просто не определены, хотя формально они входят в состав стандартных API-интерфейсов и являются важной частью механизма для работы с URL-адресами. Даже те из них, которые поставляются вместе с платформой, все равно находятся в отдельном пространстве имен `com.sun.*`, за пределами основной библиотеки.

Резюмируя все вышесказанное, механизм обработки содержимого и протоколов опередил свое время, но так и не смог реализовать до конца свой потенциал. Веб-браузеры, способные динамически расширяться для поддержки новых протоколов и типов данных, сродни летающим машинам — давно стали неотъемлемым атрибутом «ближайшего будущего». И хотя механизм работы с протоколами вполне пригоден для использования (особенно сейчас, когда его уже стандартизировали), для обработки данных вам, вероятно, стоит поискать более современное и узконаправленное решение.

Технологии обработки данных, на которые стоит обратить внимание

Идея динамической загрузки применяется и в других компонентах, предназначенных для обработки данных. Например, Java-программисты часто любят использовать формат XML в качестве кроссплатформенного средства для описания документов и предоставления функциональности. Программа для просмотра XML-файлов вполне могла бы поддерживать загрузку компонентов, предназначенных для отображения определенных тегов.

Технология JavaBeans решает эту задачу с помощью системы Java Activation Framework (JAF), которая позволяет определять тип потоковых данных и «инкапсулировать доступ к ним» посредством классов, реализующих специальный интерфейс. Очень похоже на обработчики содержимого, не так ли? К сожалению, эти два API-интерфейса, по всей

видимости, никогда не объединятся в один и не получат широкого распространения за пределами библиотек Java Mail API и JAF.

Тем не менее для потоковой передачи изображений, музыки и видеофайлов с использованием URL-адресов существуют достаточно зрелые инструменты. Библиотека Java Advanced Imaging API (JAI) включает в себя хорошо структурированный, расширяемый набор обработчиков для множества графических форматов, а технология Java Media Framework (JMF) позволяет воспроизводить большинство музыкальных и видеофайлов, которые можно найти во Всемирной паутине.

Взаимодействие с веб-приложениями

Веб-браузеры — это универсальные клиенты для веб-приложений. Они получают и отображают страницы, предоставляя пользовательский интерфейс, который состоит преимущественно из HTML, JavaScript и сопутствующих документов. В этом разделе вы научитесь писать клиентский код, который взаимодействует с веб-приложениями по протоколу HTTP. Вы будете принимать и отправлять данные, используя класс `URL` и запросы типа GET и POST. Позже в этой главе мы познакомимся с веб-сервисами, которые обеспечивают кроссплатформенное взаимодействие между приложениями посредством HTTP, XML и других веб-стандартов.

У приложения может быть множество причин для того, чтобы передавать данные по протоколу HTTP. К примеру, это гарантирует совместимость с другими приложениями на основе веб-браузеров, что может иметь большое значение, или у вас может возникнуть необходимость получить доступ к серверу, защищенному брандмауэром, прямое соединение с которым было бы проблематичным (не говоря уже об использовании технологии RMI). В Интернете стандарт HTTP является общеупотребимым, и, несмотря на свои ограничения (или, скорее, благодаря своей простоте), он очень быстро стал самым распространенным протоколом в мире. На клиентской стороне вместо веб-страниц, написанных на языке HTML, можно использовать полноценные Java-приложения (как с графическим пользовательским интерфейсом, так и без). Это позволит выполнять вывод и проверку данных любого уровня сложности, и, в то же время, использовать удаленный веб-сервис.

Прежде всего, стоит обсудить отправку данных на сервер, особенно кодирование содержимого веб-форм. В веб-браузере поля веб-форм

принимают вид «ключ/значение», кодируются в специальный формат и отправляются по сети посредством запроса. Запросы бывают двух разных типов. Первый, GET, превращает данные, введенные пользователем, в URL-адрес. Когда сервер получает такой запрос, он выделяет в нем адрес и информацию, которая была закодирована в виде параметра. Второй тип, POST, предлагает серверу принять закодированные данные и передать их веб-приложению в потоковом режиме. Платформа Java позволяет связаться с серверной программой с помощью URL-адреса и отправить ей запрос (с данными или без) типа GET или POST. (В главе 15 мы научимся создавать веб-приложения, которые работают на серверной стороне.)

Использование запросов типа GET

Кодировать информацию в URL-адресе и отправлять ее запросом типа GET довольно просто. Для этого достаточно создать объект `URL`, ссылающийся на серверное приложение, и сформировать пары «ключ/значение», из которых будут состоять наши данные. Например, следующий фрагмент кода связывается со старой программой под названием **login.cgi**, которая находится на сервере **myhost** и работает по стандарту CGI, передает ей две пары «ключ/значение» и выводит ответ на экран:

```
URL url = new URL(
   // Этот URL-адрес должен быть закодирован
   "http://myhost/cgi-bin/login.cgi?Name=Pat&Password=foobar");
BufferedReader bin = new BufferedReader (
   new InputStreamReader( url.openStream() ));
String line;
while ( (line = bin.readLine()) != null ) {
   System.out.println( line );
}
```

Чтобы было понятно, с какого именно места в URL-адресе начинаются параметры, нужно добавить после имени файла **(login.cgi)** вопросительный знак. Затем можно указать пары «ключ/значение», разделенные амперсандом (символом «&»). В оставшейся части кода мы просто открываем поток и считываем ответ, возвращаемый сервером. Не забывайте, что само по себе создание объекта `URL` не инициирует соединения. Чтобы выполнить подключение, нам пришлось вызвать метод `openStream()`. И хотя в этом примере мы ожидаем получить текст, в качестве ответа сервер может вернуть все что угодно.

Стоит отметить, что мы сознательно пропустили один этап. Этот код работает только лишь потому, что в качестве параметров был использован обычный текст. Если бы наши данные содержали «непечатные» или специальные символы (в том числе «?» и «&»), нам бы пришлось их предварительно закодировать. Для этих целей предусмотрен класс `java.net.URLEncoder`, использование которого будет продемонстрировано в следующем примере.

Есть еще один важный нюанс, о котором нужно сказать. То, как мы отправляем поле с паролем, совершенно не годится для передачи секретной информации. Наш пароль имеет вид обычного незашифрованного текста и является частью URL-адреса (то есть он может фигурировать в журнальных записях на сервере, в закладках и т. д.). О вопросах, касающихся безопасности, мы еще поговорим и в этой, и в следующей главе, когда речь пойдет о написании веб-приложений с использованием сервлетов.

Использование запросов типа POST

Ниже представлено простое приложение, которое ведет себя, как HTML-форма. Оно берет данные из двух полей ввода, `name` и `password`, и передает их по заданному URL-адресу, используя HTTP-запрос типа POST. Эта клиентская программа, основанная на библиотеке Swing, работает с серверным веб-приложением по принципу веб-браузера.

Вот ее код:

```
//файл: Post.java
import java.net.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Post extends JPanel implements ActionListener {
   JTextField nameField, passwordField;
   String postURL;
   GridBagConstraints constraints = new GridBagConstraints(   );

   void addGB( Component component, int x, int y ) {
      constraints.gridx = x;   constraints.gridy = y;
      add ( component, constraints );
   }
```

```
public Post( String postURL ) {

this.postURL = postURL;

setBorder(BorderFactory.createEmptyBorder(5, 10, 5, 5));
   JButton postButton = new JButton("Post");
   postButton.addActionListener( this );
   setLayout( new GridBagLayout(  ) );
   constraints.fill = GridBagConstraints.HORIZONTAL;
   addGB( new JLabel("Имя ", JLabel.TRAILING), 0, 0 );
   addGB( nameField = new JTextField(20), 1, 0 );
   addGB( new JLabel("Пароль ", JLabel.TRAILING), 0, 1 );
   addGB( passwordField = new JPasswordField(20), 1, 1 );
   constraints.fill = GridBagConstraints.NONE;
   constraints.gridwidth = 2;
   constraints.anchor = GridBagConstraints.EAST;
   addGB( postButton, 1, 2 );
}
public void actionPerformed(ActionEvent e) {
   postData();
}
protected void postData( ) {
   StringBuffer sb = new StringBuffer(  );
   sb.append( URLEncoder.encode("Name") + "=" );
   sb.append( URLEncoder.encode(nameField.getText(  )) );
   sb.append( "&" + URLEncoder.encode("Password") + "=" );
   sb.append( URLEncoder.encode(passwordField.getText(  )) );
   String formData = sb.toString( );
   try {
      URL url = new URL( postURL );
      HttpURLConnection urlcon =
            (HttpURLConnection) url.openConnection(  );
      urlcon.setRequestMethod("POST");
      urlcon.setRequestProperty("Content-type",
            "application/x-www-form-urlencoded");
      urlcon.setDoOutput(true);
      urlcon.setDoInput(true);
      PrintWriter pout = new PrintWriter( new OutputStreamWriter(
            urlcon.getOutputStream(  ), "8859_1"), true );
      pout.print( formData );
      pout.flush(  );
```

```
            // интерпретируем результат...
            if ( urlcon.getResponseCode(   ) == HttpURLConnection.HTTP_OK )
               System.out.println("Данные переданы!");
            else {
               System.out.println("Некорректные данные...");
               return;
            }
            //InputStream in = urlcon.getInputStream(   );
            // ...
         } catch (MalformedURLException e) {
            // некорректное значение переменной postURL
            System.out.println(e);
         } catch (IOException e2) {
            // ошибка ввода/вывода
            System.out.println(e2);
         }
      }
      public static void main( String [] args ) {
         JFrame frame = new JFrame("SimplePost");
         frame.add( new Post( args[0] ), "Center" );
         frame.pack( );
         frame.setVisible(true);
      }
   }
```

При запуске этой программы в командной строке необходимо указать URL-адрес серверного приложения. Например:

```
% java Post http://www.myserver.example/cgi-bin/login.cgi
```

Все начинается с создания окна; этот код покажется вам тривиальным после того, как вы прочитаете главы 16–18, посвященные библиотекам для построения графического пользовательского интерфейса AWT и Swing. Вся магия происходит в методе `postData()`. Сначала мы создаем буфер `StringBuffer` и загружаем в него две пары «ключ/значение», разделенные амперсандом. (При использовании запроса типа POST необходимость в предваряющем вопросительном знаке отпадает, так как мы не будем вставлять наши данные в URL-адрес.) Каждая пара предварительно кодируется с помощью метода `URLEncoder.encode()`. Несмотря на то, что в этом примере не используется никаких специальных символов, через кодировщик должны пройти как ключи, так и значения.

Далее мы подготавливаем соединение с серверной программой. В нашем предыдущем примере для отправки данных не нужно было делать ничего особенного, так как запрос всего лишь открывал URL-адрес на сервере. Здесь же нам необходимо позаботиться о некоторых аспектах сетевого взаимодействия. К счастью, большую часть работы можно переложить на объект `HttpURLConnection`; нам остается только указать тип данных, которые мы отправляем, и сообщить о том, что запрос будет иметь тип POST. Чтобы получить экземпляр `URLConnection`, мы вызываем метод `openConnection()` из объекта `URL`. Мы знаем, что здесь используется протокол HTTP, поэтому у нас должна быть возможность привести полученное значение к типу `HttpURLConnection`, который умеет делать то, что нам нужно. И поскольку данный протокол поддерживается на всех платформах, с этим не должно возникнуть никаких проблем.

Затем с помощью метода `setRequestMethod()` мы сообщаем о нашем намерении выполнить операцию POST. Мы также используем метод `setRequestProperty()`, чтобы указать для запроса HTTP-заголовок `Content-Type` — в нашем случае это будет MIME-тип закодированных данных, отправляемых через веб-форму (сервер должен знать, какие данные мы ему передаем). Напоследок мы вызываем методы `setDoOutput()` и `setDoInput()`, чтобы наше соединение могло как принимать, так и отправлять потоки информации. Зная о том, что мы собираемся выполнить операцию POST, объект `HttpURLConnection` ожидает получить какой-то ответ. Дальше, с помощью метода `getOutputStream()`, мы получаем исходящий поток и создаем на его основе объект `PrintWriter`, который позволит нам легко записать наши закодированные данные.

Отправив данные, наше приложение вызывает метод `getResponseCode()`, чтобы узнать код ответа, который был возвращен сервером, и понять, успешно ли прошел наш запрос. Любой код, кроме `HTTP_OK` (все коды хранятся в классе `HttpURLConnection` в виде констант), будет свидетельствовать о неудаче. В конце мы могли бы прочитать текст ответа, но нам достаточно знать, что отправка данных прошла успешно.

О том, что мы передаем закодированные данные через веб-форму, свидетельствует MIME-тип, указанный в заголовке `Content-Type`. И хотя это самый распространенный подход, существуют и другие способы взаимодействия с веб-приложением. Мы могли бы обмениваться данными произвольных типов, используя входящие и исходящие потоки. Запрос типа POST способен отправлять любую информацию; серверное приложение просто должно знать, как ее обрабатывать.

Напоследок стоит сказать, что для декодирования данных, отправленных через веб-форму, можно воспользоваться классом `java.net.URLDecoder`, который отменяет действие объекта `URLEncoder`. В главе 15 вы увидите, что серверы делают это автоматически.

Класс `HttpURLConnection`

Класс `HttpURLConnection` предоставляет много разной информации. С помощью методов `getContentType()` и `getContentEncoding()` можно определить MIME-тип и кодировку запроса. Мы можем также получить HTTP-заголовки ответа, используя метод `getHeaderField()` (HTTP-заголовки представляют собой метаданные в виде пар «ключ/значение»). Вспомогательные методы `getHeaderFieldInt()` и `getHeaderFieldDate()` извлекают из заголовков целые числа и даты, приводя их к типам `int` и `long` соответственно. Размер содержимого и дату последнего изменения можно узнать с помощью методов `getContentLength()` и `getLastModified()`.

Технология SSL и безопасность при взаимодействии с веб-приложениями

В предыдущем примере мы отправляли на сервер содержимое поля ввода под названием «Пароль». Но стандарт HTTP не предоставляет никаких средств шифрования, чтобы мы могли скрыть наши данные. К счастью, обезопасить запросы типа GET и POST довольно легко (для разработчика клиентского приложения это и в самом деле тривиальная задача). При любой возможности следует использовать безопасную разновидность протокола HTTP — HTTPS:

```
https://www.myserver.example/cgi-bin/login.cgi
```

HTTPS — это подмножество стандарта HTTP, которое работает поверх защищенных сокетов (Secure Sockets Layer или SSL), шифруя обмен данными между клиентом и сервером с помощью публичного ключа. Большинство современных веб-бразуеров и серверов имеют встроенную поддержку протокола HTTPS (или безопасных сокетов). Таким образом, если серверная сторона сконфигурирована для работы с HTTPS, вы можете организовать с ней безопасный обмен данными, указав в адресной строке своего веб-браузера протокол «https». Мы могли бы долго говорить о технологии SSL и других аспектах безопасности (например, об аутентификации приложений, с которыми происходит взаимодействие), но на самом деле обычного шифрования данных вполне достаточно.

Вы не будете работать с безопасными соединениями напрямую. Поддержка технологий SSL и HTTPS присутствует стандартной поставке пакета Java JRE 5.0 и выше. Все реализации платформы Java обязаны предоставлять производную класса `URLConnection` с поддержкой протоколов HTTP и HTTPS. Создание защищенных веб-приложений будет подробно рассмотрено в главе 15.

Идентификаторы URL, URN и URI

Ранее мы уже обсуждали URL-адреса, подчеркивая их отличие от идентификаторов URN. Если URL-адрес указывает на определенное место в сети и определяет протокол (или *схему*), по которому можно получить доступ к его содержимому, то URN — это просто уникальное название. Первый больше похож на номер домашнего телефона, тогда как второй представляет собой нечто вроде идентификационного номера налогоплательщика. Телефонный номер можно поменять, но идентификационный код является уникальным и выдается только один раз.

Теоретически можно создать систему, которая будет искать заданный идентификатор URN и привязывать его к определенному местоположению (URL), но на практике не все так просто. Если значение URN является постоянным, уникальным и абстрактным идентификатором элемента, то URL-адрес используется в тот момент, когда необходимо связаться с ресурсом. Иными словами, с помощью телефонного номера вы можете позвонить мне прямо сейчас, а идентификационный код всегда позволяет установить мою личность.

В качестве примера URN можно привести строку **http://www.w3.org/1999/XSL/Transform**, которая идентифицирует версию языка преобразования XML-документов (Extensible Stylesheet Language или XSL), стандартизированного консорциумом W3C. Так уж получилось, что эта строка имеет вид URL-адреса (вы можете сходить по нему, чтобы почитать об этом стандарте), но это сделано исключительно в целях удобства. Главная задача идентификатора URN — создать для определенной версии языка программирования уникальный маркер, который всегда будет оставаться неизменным.

И URN, и URL являются единообразными идентификаторами ресурсов (Uniform Resource Identifier или URI). То есть они являются подмножествами URI. Эти два вида идентификаторов по определению имеют нечто общее, поэтому они и были сгруппированы в единый стандарт.

Все идентификаторы URI должны легко восприниматься и воспроизводиться человеком (так, чтобы вы могли записать их на обратной стороне салфетки). Они всегда имеют иерархическую структуру и являются уникальными. Синтаксис идентификаторов URL и URN, описанный в спецификации RFC 2396, частично совпадает.

Класс `java.net.URI` формализует различия между этими идентификаторами. В отличие от `URL`, он не пытается разбивать идентификатор на отдельные элементы или придавать им какой-то «смысл». Если класс `URL` сразу же вычленяет из адреса протокол и ищет для него подходящий обработчик, то `URI` вообще никак не интерпретирует содержимое идентификатора. Он нужен лишь для того, чтобы мы могли работать с идентификатором, как со структурированным текстом, учитывая общие положения стандарта URI. Класс `URI` позволяет сформировать строку, получить относительный путь и проверить значения на равенство, но он ничего не знает о протоколе или имени сервера.

Веб-сервисы

Это обширная и стремительно развивающаяся сфера программирования, которой посвящено множество других прекрасных книг. Но мы уже успели охватить так много основных сетевых концепций (а в главе 24 мы еще возьмемся за формат XML), что игнорировать эту важную отрасль разработки приложений было бы с нашей стороны безответственно. Данный раздел, посвященный клиентской стороне сетевого взаимодействия, завершается небольшим примером удаленного использования веб-сервиса.

В отличие от других веб-приложений, рассчитанных на работу через веб-браузер, веб-сервисы представляют собой API-интерфейсы, обращаться к которым должны другие программные компоненты. Чтобы улучшить переносимость и доступность, они используют веб-стандарты и формат XML — это их главное отличие от других механизмов межпрограммного взаимодействия. Мы не станем здесь обсуждать, в каких случаях такой подход оказывается целесообразным, и на какие компромиссы ради него придется пойти. О ценности данной идеи можно судить хотя бы по взрывообразному росту популярности прикладных веб-приложений, который мы наблюдаем в последние годы. Веб-сервисы предоставляют хорошо структурированные, кроссплатформенные интерфейсы, с помощью которых программы, основанные на веб-технологиях, могут общаться друг с другом.

Технология XML-RPC

Термин «*веб-сервис*» можно понимать совершенно по-разному. За последние годы появилось столько новых веб-стандартов с префиксом «WS» (от «Web-Service»), что для них даже придумали общее обозначение — «WS-*» (иногда эту звездочку иронично называют «звездой смерти»). Но изначальная концепция довольная простая: мы берем широко распространенную, понятную и легко реализуемую операцию, которая выполняется поверх протокола HTTP, и с помощью формата XML описываем, как ее можно вызывать. Это разновидность *удаленного вызова процедур* (Remote Procedure Call или RPC), в которой протокол HTTP играет привычную для себя роль базового механизма для обмена данными, а формат XML предоставляет «прикладную оболочку» внутри которой эти данные структурируются и передаются. Веб-сервис, написанный в стиле технологии RPC, определяет базовую структуру вызова (запроса) и формат упаковки для простых типов данных, обеспечивая настоящую кроссплатформенность параметров и возвращаемых результатов. Существуют также веб-сервисы «документного типа», в которых основной акцент делается на обмен XML-документами между приложениями, а не на упаковку и распаковку данных, как в RPC. На сегодняшний день самой тесной интеграцией с платформой Java обладают веб-сервисы, основанные на удаленном вызове процедур, поэтому мы сосредоточимся именно на них.

Язык WSDL

Ключевым в этом контексте является язык для описания веб-сервисов (Web Services Description Language или WSDL). Этот стандарт позволяет описать в формате XML сам веб-сервис, отдельные функции (методы), которые он предоставляет, а также типы данных для аргументов и возвращаемых значений. Технология WSDL является ответвлением языка IDL (Interface Definition Language), и ее главная задача — описывать интерфейсы. Но в документе формата WSDL можно также указать местоположение и другие свойства, которые обычно не имеют прямого отношения к веб-сервисам.

С точки зрения разработчика клиентских приложений документ формата WSDL содержит все сведения, необходимые для вызова методов веб-сервиса (с использованием любого языка, включая Java). С его помощью вы можете сгенерировать клиентские интерфейсы и исполь-

зовать их в своем Java-приложении. В главе 15 вы научитесь выполнять обратную операцию: генерировать WSDL-документ для нового веб-сервиса на основе своего собственного Java-кода.

Инструменты для работы с веб-сервисами

Технология JAX-WS (Java API for XML Web Services) входит в состав платформы Java начиная с версии 6 и содержит все инструменты, необходимые для создания и использования веб-сервисов. Она также предоставляет стандартные средства для развертывания веб-сервисов с целью их дальнейшего тестирования. Для работы с документами формата XML предусмотрен набор API-интерфейсов под названием JAXP (Java API for XML Processing). Технология JAX-WS может предложить классы для осуществления удаленных вызовов, а также утилиты `wsimport` и `wsgen`, которые могут пригодиться на этапе разработки. Программа `wsimport` считывает WSDL-файл и генерирует на его основе интерфейсы и соответствующие классы-реализации, необходимые для взаимодействия с веб-сервисом. Программа *wsgen* анализирует код на языке Java, который содержит аннотации, касающиеся использования веб-сервиса, и генерирует описание в формате WSDL, а также другие файлы, связанные с процессом развертывания.

Существует множество серверов приложений, которые предоставляют собственные механизмы для развертывания веб-сервисов и генерирования клиентского кода. В качестве примера можно привести популярный проект Apache CXF, который поддерживает JAX-WS и другие стандарты.

Клиент для метеослужбы

В этом примере мы покажем, насколько просто использовать веб-сервисы на клиентской стороне. Мы создадим клиент для удаленной метеослужбы. Сервис принимает в качестве аргумента почтовый индекс (к сожалению, только американский) и возвращает город, штат и погодные условия. Обратите внимание на то, что серверная сторона этого примера предоставляется компанией **cdyne.com** — промышленным поставщиком веб-сервисов. Поскольку это сторонний веб-сайт, мы не знаем, сможете ли вы его открыть, когда будете читать эту книгу. Если по какой-то причине он исчезнет, не волнуйтесь — в главе 15 мы создадим свой собственный веб-сервис.

Для начала нам нужно получить файл формата WSDL с описанием веб-сервиса. Вы можете найти его прямо на одной из страниц веб-сайта **cdyne.com**. Это XML-документ, который определяет набор операций и типов данных, которые будут использоваться для аргументов и возвращаемых результатов.

Этот файл не рассчитан на то, что его будет читать живой человек, но после знакомства с форматом XML в главе 24 вы будете лучше понимать его содержимое.

Чтобы сгенерировать клиентский код, необходимый для взаимодействия с веб-сервисом, нужно запустить утилиту `wsimport`, которая находится в каталоге *bin* пакета JDK, и передать ей местоположение WSDL-файла:

```
% wsimport http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL
```

Когда утилита `wsimport` завершит свою работу, у вас должно будет появиться новое дерево каталогов вида **com/cdyne/ws/weatherws** со скомпилированными *class*-файлами внутри; там вы найдете интерфейс и реализацию клиентской части взаимодействия с метеослужбой. У команды `wsimport` есть множество полезных опций: ключ `-keep` позволит сохранить исходный код клиентских классов, чтобы вы могли хранить его вместе со своим приложением. Есть также ключ `-p`, который позволяет переопределить название генерируемого пакета.

Сгенерированный код содержит класс под названием `Weather`, который представляет сервис как таковой, и интерфейс `WeatherSoap`, предоставляющий наборы методов для различных реализаций (в формате WSDL наборы методов обозначаются тегом `port`). Если сохранить исходный код (с помощью ключа `-keep`) и просмотреть его содержимое, можно увидеть, что элементы веб-сервиса идентифицируются с помощью стандартных для языка Java аннотаций. Класс `Weather` и интерфейс `WeatherSoap` помечены как `@WebServiceClient` и `@WebService` соответственно. Для методов интерфейса `WeatherSoap` тоже указаны аннотации (`@WebMethod`). Эти метаданные позволяют идентифицировать веб-сервис и взаимодействовать с ним с помощью формата XML, используя информацию из WSDL-файла. Более подробно об аннотациях мы поговорим в следующей главе, когда приступим к созданию и развертыванию серверной части нашего веб-сервиса. Аналогичное применение аннотаций можно будет встретить в главе 24, когда речь пойдет о связывании данных посредством формата XML и технологии JAXB.

Теперь наше клиентское приложение может использовать эти классы для общения с веб-сервисом. Следующий код запрашивает текущую погоду для населенного пункта с почтовым индексом 02645:

```
import com.cdyne.ws.weatherws.*;
public class WSTest {
   public static void main( String[] args )
   {
      WeatherSoap weatherService = new Weather().getWeatherSoap();
      WeatherReturn weather = weatherService.getCityWeatherByZIP( "02645" );
      System.out.format("%s, %s : %s : Температура: %s, Ветер: %s",
         weather.getCity(), weather.getState(), weather.getDescription(),
         weather.getTemperature(), weather.getWind() );
   }
}
```

Не забывайте, чтобы пример заработал, вам необходимо либо добавить class-файлы в директорию classpath, либо скомпилировать сгенерированный исходный код вместе со своим приложением. Ниже показан результат выполнения этого примера. Обратите внимание: несмотря на то, что в нашем случае значения передаются в виде строк, большинство WSDL-интерфейсов позволяют использовать числовые типы для таких элементов как температура.

```
Harwich, MA : Sunny : Температура : 37, Ветер: CALM
```

В следующей главе мы еще вернемся к этой теме. Переметнувшись на серверную сторону, мы приступим к разработке веб-приложений и, наконец, создадим свой собственный веб-сервис.

Материал, защищенный авторским правом

Глава 15

ВЕБ-ПРИЛОЖЕНИЯ И ВЕБ-СЕРВИСЫ

Чтобы научиться создавать веб-приложения и веб-сервисы, мы перейдем от клиентской стороны к серверной. В отличие от обычных Java-программ, веб-приложения хранят большую часть своего кода, бизнес-логики и данных на сервере (по крайней мере, изначально), и для доступа к ним необходимо использовать веб-браузер или легковесный клиент. Изящности данного подхода к разработке программного обеспечения способствует улучшенная поддержка стандартов в современных веб-браузерах, мощь языков HTML и JavaScript, высокоскоростной доступ к Интернету, а также усовершенствованные средства для прямых соединений между приложениями.

Большая часть этой главы посвящена Servlet API — платформе, предназначенной для написания серверных программных компонентов. Эта платформа часто используется как в обычных веб-приложениях, так и в веб-сервисах, обеспечивающих межпрограммное взаимодействие. Непосредственная работа с сервлетами начнется уже в первом разделе этой главы; мы напишем несколько примеров, которые можно использовать через веб-браузер. Позже мы рассмотрим веб-сервисы, которые работают в более «скрытной» манере и способны предоставлять данные и услуги для любых типов клиентских приложений. Эти два вида серверных компонентов имеют кое-что общее — например, они развертываются на сервере приложений в виде файлов в формате WAR (Web Application Archive) и часто используются вместе внутри сложных программных систем, которые генерируют страницы на серверной стороне и передают данные клиенту с помощью языка JavaScript.

API-интерфейсы для работы с сервлетами находятся в пакете `javax.servlet`, который является расширением стандартной библиотеки. Развертывание и запуск сервлетов требуют наличия *сервера приложений* (или *контейнера сервлетов*) — сервера, написанного на языке Java, который играет роль веб-сервера и обрабатывает запросы, привязанные к компонентам сервлета. Таким образом, интерфейсы из состава Servlet API не входят в стандартную поставку платформы Java. Для запуска

примеров, которые рассматриваются в этой главе, мы рекомендуем вам загрузить сервер Apache Tomcat; также, чтобы компилировать классы из этих примеров, вам понадобится JAR-файл с API-интерфейсами для работы с сервлетами. Некоторые интегрированные среды разработки могут установить нужный архив автоматически.

API-интерфейсы, с помощью которых создаются и развертываются веб-сервисы для межпрограммного взаимодействия, являются частью пакета `javax.jws`. И хотя формально технология JWS API считается расширением, она входит в состав стандартной платформы Java, поэтому вы можете создавать веб-сервисы сразу, не загружая дополнительных компонентов. Более того, в стандартной поставке пакета JDK содержатся минималистические средства для развертывания веб-сервисов, так что использовать отдельный сервер не обязательно. Но нужно понимать, что эти средства в первую очередь предназначены для тестирования и не могут рассматриваться в качестве полноценной замены настоящих серверов приложений, таких как Tomcat. В данной главе рассматриваются технологии Java Servlet API 3.0 и JWS (JAX-WS) версии 2.2.

Серверы, которые поддерживают полный набор компонентов Java Enterprise API, включая сервлеты, веб-сервисы, JSP-сценарии и более старые технологии, такие как Enterprise JavaBeans, называют *серверами приложений*. Примерами таких серверов могут служить открытый проект JBoss и его популярный коммерческий аналог от компании BEA System — WebLogic. Пакет Apache Tomcat, который мы будем использовать в этой главе, изначально создавался как контейнер сервлетов, но со временем обзавелся поддержкой веб-сервисов и других компонентов, необходимых для разработки серьезного программного обеспечения. Tomcat может использоваться как отдельно, так и в связке с другим веб-сервером наподобие Apache. Это простое в настройке приложение полностью написано на языке Java, поэтому вы можете использовать его на любом компьютере с установленной Java-машиной. Загрузить его можно на странице **jakarta.apache.org/tomcat/**.

Технологии, лежащие в основе веб-приложений

Инструменты для создания серверных веб-приложений постоянно эволюционируют. Когда-то в качестве стандарта выступала технология CGI, которая позволяла обслуживать запросы, отправляемые из веб-браузеров, с помощью скриптовых языков программирования, таких

как Perl. Некоторые веб-серверы предлагали API-интерфейсы, написанные на компилируемых языках; например, модули для веб-сервера Apache написаны на C и C++. Однако технология Java Servlet API, обладая переносимостью, безопасностью и высокой производительностью, быстро стала самой популярной архитектурой для построения веб-приложений. В наши дни существует множество инструментов для создания веб-сервисов; системы, написанные на языке Java, конкурируют с аналогичными решениями на базе Microsoft .NET и альтернативными платформами, такими как Ruby on Rails. Однако современные тенденции таковы, что главный приоритет отдается клиентским технологиям вроде JavaScript и HTML5, поэтому не так уж важно, на каком языке реализованы серверные компоненты или веб-сервисы. Мы постараемся помочь вам сориентироваться в этом изобилии технологий.

Многостраничные и одностраничные приложения

Большую часть того времени, что существует платформа Java, веб-приложения работали по одному и тому же простому принципу: браузер делает запрос по определенному URL-адресу, а сервер генерирует ответ в виде HTML-документа. При этом любые действия пользователя приводили к переходу на следующую страницу. В этом процессе вся работа (или большая ее часть) выполнялась серверной стороной, что, на первый взгляд, выглядит вполне логично, учитывая то, где именно находятся данные и сам веб-сервис. Однако у такой архитектуры есть свои естественные ограничения, выраженные в нехватке отзывчивости и отсутствии целостности. Сложно добиться от веб-приложения такой же плавной работы, как в настольных программах, если пользователь вынужден «перепрыгивать» от одной страницы к другой, каждый раз ожидая загрузку в браузере. К тому же возникает проблема сохранения данных между этими переходами. В конце концов, веб-браузеры создавались для просмотра документов, а не для выполнения полноценных программ.

Но за последние годы в мире веб-разработки произошли большие перемены. Стандарты HTML и JavaScript вышли на тот уровень, когда большую часть пользовательского интерфейса и бизнес-логики можно хранить на клиентской стороне, запрашивая у сервера данные и различные функции в фоновом режиме. Согласно этой концепции сервер должен вернуть всего одну «страницу» формата HTML, в которой содержатся ссылки на скрипты, таблицы стилей и другие ресурсы, необходимые для генерирования интерфейса приложения. Дальше эста-

фету принимают скрипты, написанные на языке JavaScript: они могут динамически создавать новые элементы страницы и управлять уже существующими, формируя пользовательский интерфейс с помощью богатых возможностей технологии HTML DOM. Для получения данных и вызова удаленных функций применяются асинхронные (фоновые) запросы. Часто результат возвращается в формате XML, благодаря чему этот вид взаимодействия получил название AJAX (Asynchronous Javascript and XML — асинхронный JavaScript и XML).

Благодаря этому новому подходу многие аспекты веб-разработки значительно упростились. Больше нет никакой необходимости работать в «постраничном» режиме, обмениваясь с сервером запросами и ответами. Теперь клиент больше похож на настольное приложение — он может мгновенно реагировать на действия пользователя и управлять удаленными данными и функциями, не прерывая своей работы. Прежде чем перейти к сервлетам, давайте быстро пройдемся по старым и новым веб-технологиям, которые имеют отношение к платформе Java.

JSP-сценарии

JSP — это еще одна технология для создания серверных приложений, ориентированных, в основном, на вывод отдельных страниц (документов). Такие приложения состоят из HTML-кода, внутри которого поддерживаются нестандартные теги и синтаксис языка Java. JSP-сценарии динамически компилируются веб-сервером, в результате чего получаются сервлеты, которые генерируют содержимое страниц и способны работать с Java-интерфейсами как прямо, так и опосредовано. И хотя весь код выполняется на серверной стороне, все выглядит так, будто бизнес-логика находится прямо внутри страницы. У такого подхода есть свои преимущества и недостатки. Преимущество заключается в том, что стиль программирования, при котором разработчик имеет дело непосредственно с HTML-документом, довольно прост для понимания и может быть легко поставлен на поток. Но отсюда следует и недостаток: страницы превращаются в запутанную смесь из бизнес-логики и кода для отображения пользовательского интерфейса. Чем сильнее код смешивается со статической разметкой, тем сложнее становится его поддерживать.

Большинство крупномасштабных проектов, основанных на технологии JSP, используют библиотеки тегов, чтобы уменьшить количество лишнего кода. При написании JSP-сценариев все сложные операции и бизнес-логику часто выносят в отдельный *контроллер* (который тоже

является сервлетом). В этом случае применяется архитектура «модель-представление-контроллер» (Model-View-Controller или MVC), когда каждая задача решается с помощью отдельных компонентов (мы уже упоминали об этом при обсуждении пользовательских интерфейсов на основе библиотеки Swing). Такое разделение нивелирует недостатки технологии JSP, позволяя в полной мере воспользоваться ее преимуществами.

XML и XSL

XML — это набор стандартов для работы со структурированной информацией, представленной в виде текста. XSL (Extensible Stylesheet Language) — это язык для преобразования XML-документов в другие форматы, в том числе и в HTML. Сочетание сервлетов, которые могут генерировать XML-код, и XSL-таблиц, способных подготавливать содержимое для отображения, обладает большими возможностями и будет подробно рассмотрено в главе 24. Как вы позже увидите, веб-сервисы используют XML в качестве основного формата данных, благодаря чему они совершенно не привязаны к конкретным платформам и языкам. И, конечно же, стоит сказать, что XML является базовым форматом для обмена информацией с AJAX-приложениями, написанными на языке JavaScript.

Фреймворки для создания веб-приложений

Если рассматривать веб-приложения в контексте классической архитектуры MVC, то «представление» в традиционной «многостраничной» модели генерируется в рамках веб-браузера, а модель (данные) и контроллер (бизнес-логика) находятся на серверной стороне. Мы уже говорили, почему веб-приложения все чаще используют «одностраничную» концепцию, где в браузер выносится больше компонентов; тем не менее, за эти годы было создано множество фреймворков, которые до сих пор поддерживают классический подход. Чаще всего такие фреймворки работают на более высоком уровне в сравнении с сервлетами; они предоставляют удобные средства для написания и подключения контроллеров, позволяя конфигурировать страницы для вывода результатов.

В свое время одним из самых популярных инструментов для построения многостраничных веб-приложений был фреймворк Apache Struts Web Application Framework. Поддерживая парадигму MVC, он имеет модульную структуру контроллеров и расширяемую библиотеку тегов для представлений, в качестве которых выступают JSP-сценарии.

Фреймворк Struts совмещает в себе некоторые аспекты навигации и распределения вызовов, благодаря которым компоненты объединяются в единое веб-приложение. Он использует конфигурационные файлы в формате XML и позволяет применять декларативный стиль для связывания HTML-форм с объектами языка Java, а также выполнять автоматическую проверку полей ввода.

Ответом компании Sun на фреймворк Struts стал проект JSF. Он был разработан в рамках инициативы Java Community Process (с участием некоторых создателей Struts) и должен был стать «официально одобренным» фреймворком для веб-приложений на языке Java. Он получил усовершенствованную модель MVC с серверными программными компонентами, а также более тонкое управление навигацией и событиями. Обладая довольно противоречивой репутацией, он так никогда и не превзошел в популярности своего конкурента, Struts.

Еще одна популярная система для создания веб-приложений с архитектурой MVC, Spring Web Flow, основана на фреймворке Spring. И таких примеров очень и очень много.

Google Web Toolkit

Google Web Toolkit (GWT) — это открытый и бесплатный фреймворк от компании Google. С его помощью разработчики могут компилировать свой Java-код в скрипты на языке JavaScript, которые выполняются в веб-браузере и взаимодействуют с сервером посредством специального механизма, основанного на технологии RPC (внешне это очень похоже на интерфейсы RMI). Среда выполнения фреймворка GWT предоставляет свой собственный набор классов для создания пользовательского интерфейса, а также богатый набор стандартных библиотек. Эта мощная платформа позволяет разрабатывать большие и сложные приложения; благодаря ей вы можете использовать большую часть преимуществ языка Java на стороне веб-браузера. Однако в сравнении с другими фреймворками GWT имеет довольно высокий порог вхождения (особенно для тех, кто не знаком ни с Java, ни с JavaScript).

HTML5, AJAX и т. д.

Платформа Java обитает на серверной стороне веб-приложения. Для создания клиентской части, которая будет работать в веб-браузере, мы

вынуждены воспользоваться услугами далекого родственника языка Java — JavaScript. Как мы уже упоминали, попытки стандартизировать продвинутые возможности технологий HTML и JavaScript, предпринятые в последние годы, произвели настоящую революцию в разработке клиентских веб-приложений, значительно улучшив их производительность. По большому счету, начало этой революции положила технология AJAX, которая сделала поведение клиентской части более динамическим. Взрывной рост популярности мобильных устройств, начавшийся относительно недавно, ускорил внедрение стандарта HTML5; благодаря этому веб-браузеры обзавелись более полной объектной моделью, поддержкой аудио- и видеоформатов, стандартными средствами для рисования, инструментами для работы с векторной графикой и автономным хранилищем данных. По мере продвижения процесса стандартизации мы получаем все новые и более захватывающие возможности. К примеру, веб-сокеты (WebSockets), которые позволяют обмениваться данными между веб-браузером и сервером с минимальной задержкой, должны породить массу совершенно новых приложений.

Веб-приложения на платформе Java

Пока что мы употребляли термин «веб-приложение» в общем смысле, имея в виду любую серверную программу, доступ к которой можно получить из веб-браузера. Пришло время внести больше конкретики. В контексте технологии Java Servlet API веб-приложение представляет собой набор сервлетов и веб-сервисов, которые поддерживают классы, написанные на языке Java, и могут работать с такими ресурсами, как HTML- и JSP-страницы, изображения и конфигурационная информация. Чтобы развернуть (установить и запустить) веб-приложение на сервере, его упаковывают в WAR-файл. Этот формат мы рассмотрим немного позже, но сейчас вам достаточно знать, что это обычный JAR-архив, который помимо программных файлов содержит некоторые данные, необходимые для установки. WAR-файлы не только обеспечивают переносимость приложений, но и предоставляют стандартный способ для их развертывания на сервере.

Большинство архивов формата WAR имеют собственный конфигурационный файл под названием **web.xml**. Он содержит данные о сервлетах, которые нужно развернуть, их имена, пути, параметры для инициализации и массу другой информации, включая требования к безопасности и аутентификации. Однако в последние годы многие приложения стали

игнорировать этот файл и использовать вместо него аннотации. В большинстве случаев для развертывания сервлетов и веб-сервисов достаточно пометить свои классы, предоставив всю необходимую информацию, или просто упаковать их в WAR-архив (можно сделать и то и другое). Подробней об этом мы поговорим позже в этой главе.

Веб-приложения выполняются в определенной среде. У каждого из них есть свой «корневой» путь на веб-сервере; это означает, что все URL-адреса, указывающие на сервлеты и файлы, имеют общий уникальный префикс (например, **http://www.example.com/someapplication/**). Веб-приложения изолированы друг от друга и имеют доступ только к своим файлам (хотя, конечно же, они могут обойти это ограничение с помощью веб-сервера). Каждое приложение обладает собственным *контекстом* — общим местом, где сервлеты могут обмениваться информацией и получать ресурсы из среды выполнения (мы еще к этому вернемся). Благодаря высокой степени изоляции современные промышленные системы могут динамически развертываться и обновляться, не имея никаких проблем с надежностью и безопасностью. Действительно, веб-приложения должны быть автономными и относительно самодостаточными, не связанными между собой. И хотя ничего не мешает вам наладить тесное межпрограммное взаимодействие на высоком уровне, для этих целей лучше использовать веб-сервисы (их мы обсудим позже в этой главе).

Жизненный цикл сервлета

Давайте теперь перейдем к технологии Servlet API и начнем наконец создавать сервлеты. Позже мы заполним все пробелы, когда вернемся к обсуждению структуры WAR-файлов и различных аспектов API-интерфейсов. Итак, технология Servlet API очень простая (чем-то напоминает Applet API). Базовый класс, `Servlet`, предоставляет три метода для управления жизненным циклом — `init()`, `service()` и `destroy()`; он также содержит средства для получения конфигурационных параметров и ресурсов сервлета. Однако нужно понимать, что эти методы не так уж часто вызываются вручную. Обычно разработчики реализовывают операции `doGet()` и `doPost()` для дочернего класса `HttpServlet`, и затем получают доступ к общим ресурсам через контекст сервлета, о чем мы вскоре поговорим.

Обычно в контейнере может быть запущен только один экземпляр класса `Servlet`. Если точнее, один экземпляр для каждой записи в файле **web.xml** (развертывание сервлетов мы обсудим позже). Раньше это пра-

вило можно было обойти с помощью специальных однопоточных сервлетов на основе класса `SingleThreadModel`. Но с выходом Servlet API 2.4 этот класс был признан устаревшим.

По умолчанию сервлеты должны обрабатывать запросы в многопоточном режиме; то есть их метод `service()` может запускаться одновременно в нескольких потоках. Это означает, что вам не стоит хранить в полях объекта `Servlet` данные, которые относятся к конкретному запросу или клиенту (туда можно записывать общую информацию, связанную с работой сервлета, если она не зависит от поступающих запросов). Для хранения информации между клиентскими запросами существуют серверные *сессии* и клиентские файлы cookie. Отслеживание состояния на стороне клиента тоже будет рассмотрено немного позже.

Метод `service()` из класса `Servlet` принимает два объекта: «запрос» и «ответ». Это дает возможность считывать запрос клиента и генерировать соответствующий вывод; данные объекты (или, скорее, их версии из класса `HttpServlet`) будут подробно рассмотрены в последующих примерах.

Сервлеты

Прежде всего, нас интересует пакет `javax.servlet.http`; он содержит API-интерфейсы для создания сервлетов, которые способны обрабатывать HTTP-запросы, поступающие на веб-сервер. Теоретически вы можете разрабатывать сервлеты и для других протоколов, но так как в реальности этого никто не делает, мы будем считать, что все они работают со стандартом HTTP.

В основе пакета `javax.servlet.http` лежит базовый класс `HttpServlet`. Это абстрактный сервлет, который реализует некоторые основные аспекты обработки HTTP-запросов. В частности, он переопределяет универсальный метод `service()`, разбивая его на несколько дополнительных операций, включая `doGet()`, `doPost()`, `doPut()` и `doDelete()`. Стандартный метод `service()` анализирует тип запроса и делает для него один из вышеперечисленных вызовов; вы можете переопределять их по своему усмотрению, реализовывая интересующие вас особенности протокола.

Методы `doGet()` и `doPost()` соответствуют стандартным HTTP-запросам типа GET и POST. Операция GET предназначена для получения файла или документа по заданному URL-адресу. С помощью операции POST клиент отправляет на сервер данные произвольного объема.

Именно запросы этого типа используются в HTML-формах и большинстве веб-сервисов.

Стоит также упомянуть методы `doPut()` и `doDelete()`, которые соответствуют менее распространенным HTTP-запросам, предназначенным для загрузки и удаления файлов (и похожих элементов). Операция, которую выполняет метод `doPut()`, похожа на POST, но имеет немного другое предназначение (запрос PUT должен производить замену элемента с соответствующим идентификатором, тогда как POST просто передает данные по заданному адресу). У метода `doDelete()` противоположная задача — он выполняет удаление.

Класс `HttpServlet` также реализует три других метода, связанных с протоколом HTTP: `doHead()`, `doTrace()` и `doOptions()`. Обычно их не нужно переопределять. Метод `doHead()` выполняет команду HEAD, которая запрашивает у сервера исключительно заголовки, без тела ответа. В классе `HttpServlet` этот метод выполнен в довольно примитивной манере: он делает обычный запрос типа GET, а потом отбрасывает все лишнее. Вы можете реализовать свою, более эффективную версию, если вам нужна такая оптимизация. Методы `doTrace()` и `doOptions()`, которые тоже реализуют некоторые возможности протокола HTTP, предназначены для отладки и выполнения простейшего взаимодействия между клиентом и сервером. Обычно их не стоит переопределять.

Помимо класса `HttpServlet` пакет `javax.servlet.http` содержит производные объектов `ServletRequest/HttpServletRequest` и `ServletResponse/HttpServletResPonse`, которые предоставляют, соответственно, входящие и исходящие потоки, необходимые для чтения и записи данных на клиентской стороне. Кроме того, они поддерживают API-интерфейсы для получения и указания информации об HTTP-заголовках, а также, как мы скоро увидим, средства для работы с сессиями. Вместо того чтобы начинать скучное, формальное описание, давайте рассмотрим их в контексте некоторых примеров. Как обычно, начнем с самого простого.

Сервлет `HelloClient`

Возьмем наше старое приложение «Привет, мир!» и сделаем из него сервлет:

```
@WebServlet(urlPatterns={"/hello"})
public class HelloClient extends HttpServlet
{
```

```
   public void doGet(HttpServletRequest request, HttpServletResponse
response)
      throws ServletException, IOException
   {
      // эта строка должна идти первой
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println(
         "<html><head><title>Привет, клиент!</title></head><body>"
         + "<h1>Привет, клиент!</h1>"
         + "</body></html>" );
   }
```

Если вы хотите попробовать этот сервлет в работе, можете сразу переходить к его пошаговой установке, которая рассматривается в разделе «WAR-файлы и развертывание веб-приложений» далее в этой книге. Поскольку мы добавили в наш класс аннотацию `WebServlet`, развертывание этого сервлета можно выполнить без файла *web.xml*. Достаточно просто поместить *class*-файл в определенный каталог внутри контейнера WAR (который, напомним, является обычным ZIP-архивом) и затем разместить все это в директории, которая отслеживается сервером Tomcat. Но пока что мы сосредоточимся на коде этого довольно тривиального сервлета.

Давайте рассмотрим наш пример. `HelloClient` является наследником класса `HttpServlet` и переопределяет метод `doGet()`, чтобы иметь возможность обрабатывать простые запросы. В данном случае мы хотим отвечать на любые команды типа GET, отправляя обратно однострочный HTML-документ с сообщением «Привет, клиент!». Сначала мы сообщаем контейнеру тип ответа, который мы собираемся сгенерировать; для этого используется метод `setContentType()` из объекта `HttpServletResponse`. В качестве MIME-типа указываем строку «`text/html`». Затем с помощью метода `getWriter()` мы получаем исходящий поток и выводим в него сообщение. После этого его можно не закрывать. В данной главе мы еще не раз коснемся темы управления исходящими потоками.

Исключения `ServletException`

В нашем примере в определении метода `doGet()` указано исключение `ServletException`. То же самое сделано для всех разновидностей метода `service()` на случай, если обработка запроса завершится неудачно. При создании исключения `ServletException` можно указать текстовое

сообщение и, опционально, параметр Throwable, который может позаботиться о любых сопутствующих исключениях, лежащих в основе возникшей проблемы:

```
throw new ServletException("Полный провал", someException );
```

По умолчанию веб-сервер сам определяет, что именно увидит пользователь при возникновении исключения ServletException; часто разработчику доступен так называемый «отладочный режим», в котором выводится как само исключение, так и весь его стек. С помощью файла *web.xml* можно назначать свои собственные страницы для вывода ошибок (см. раздел «Вывод ошибок и индексные страницы» далее в главе).

Сервлет также может сообщить о том, что ему не удалось обработать запрос; для этого у исключения ServletException есть производная под названием UnavailableException. Эту ошибку можно сделать постоянной или указать время (в секундах), на протяжении которого она будет повторяться.

Тип содержимого

Прежде чем получить исходящий поток и выполнить в него запись, мы должны указать тип данных, которые мы отправляем. Для этого из параметра response необходимо вызвать метод setContentType(). В нашем примере мы указали значение «text/html», которое является подходящим MIME-типом для HTML-документа. Но сервлет может сгенерировать данные любого типа, включая аудио, видео и другие разновидности текстовых и двоичных файлов. Если бы мы разрабатывали универсальный сервлет FileServlet, который обслуживает файлы по аналогии с обычным веб-сервером, мы могли бы определять MIME-тип по расширениям файловых имен или путем анализа самих данных (здесь бы нам пригодился метод probeConentType() из класса java.nio.file.Files). Для записи двоичных данных вы можете использовать метод getOutputStream(), чтобы вместо объекта Writer получить поток OutputStream.

Тип содержимого используется в заголовке «Content-Type», когда сервер отправляет свой ответ по протоколу HTTP; благодаря этому клиент знает, чего ожидать, еще до того, как начнется считывание результата. Это позволяет вашему веб-браузеру выводить подходящее диалоговое окно для сохранения файла, когда вы щелкаете мышью по ZIP-архиву или исполняемой программе. Если вы укажете заголовок «Content-Type» в полном формате (например, «text/html;

charset=ISO-8859-1»), сервлет автоматически установит соответствующую кодировку символов для исходящего потока PrintWriter. Таким образом, прежде чем получать исходящий поток с помощью метода getWriter(), вы обязательно должны сделать вызов setContentType(). Кодировку символов можно задать отдельно, вызвав из параметра response метод setCharacterEncoding().

Ответ сервлета

Помимо предоставления исходящего потока для передачи данных клиенту, объект HttpServletResponse содержит методы для управления другими аспектами HTTP-ответа, такими как заголовки, коды ошибок, перенаправления и буферизация на уровне контейнера.

HTTP-заголовки, которые представляют собой метаданные в формате «ключ/значение», отправляются вместе с ответом. Вы можете добавлять стандартные и нестандартные заголовки с помощью методов setHeader() и addHeader() (у одного ключа может быть несколько значений). Есть также вспомогательные методы для передачи целых чисел и дат:

```
response.setIntHeader("MagicNumber", 42);
response.setDateHeader("CurrentTime", System.currentTimeMillis() );
```

Когда вы передаете данные клиенту, контейнер сервлета автоматически указывает значение «200» в качестве кода ответа; это означает, что все прошло хорошо. С помощью метода sendError() можно сгенерировать другие стандартные коды, значения которых хранятся в классе HttpServletResponse в виде констант. Вот несколько наиболее распространенных из них:

```
HttpServletResponse.SC_OK
HttpServletResponse.SC_BAD_REQUEST
HttpServletResponse.SC_FORBIDDEN
HttpServletResponse.SC_NOT_FOUND
HttpServletResponse.SC_INTERNAL_SERVER_ERROR
HttpServletResponse.SC_NOT_IMPLEMENTED
HttpServletResponse.SC_SERVICE_UNAVAILABLE
```

После того как вы генерируете ошибку, вызвав метод sendError(), ответ завершится, и вы больше не сможете передавать данные клиенту. Но вы можете указать короткое сообщение, которое будет выведено на клиентской стороне (см. раздел «Простой фильтр» далее в главе).

Перенаправление — это специальный вид HTTP-ответа, который заставляет веб-браузер перейти по другому URL-адресу. Обычно это происходит быстро, без малейшего участия со стороны пользователя. Выполнить перенаправление можно с помощью метода `sendRedirect()`:

```
response.sendRedirect("http://www.example.com/");
```

Говоря об ответе сервера, мы не можем не упомянуть о буферизации. В большинстве случаев данные буферизируются контейнером сервлета до тех пор, пока метод `service()` не закончит свою работу или не будет достигнут максимальный размер буфера, определенный заранее. Это позволяет контейнеру автоматически устанавливать HTTP-заголовок «`content-length`», по которому клиент может понять, сколько данных должно поступить. С помощью метода `setBufferSize()` можно задать размер буфера (в байтах). Пока на клиентскую сторону не было ничего передано, буфер можно очищать и наполнять заново. Для этого нужно последовательно вызвать методы `isCommitted()` (чтобы проверить, не записывались ли данные в исходящий поток) и `resetBuffer()`. Размер передаваемого содержимого можно указать вручную, используя метод `setContentLength()`; это актуально, если речь идет о больших объемах данных.

Параметры сервлета

В нашем первом примере мы показали, как принимается простейший запрос. Конечно, чтобы сделать нечто действительно полезное, нам нужно будет получить от клиента какую-нибудь информацию. К счастью, сервлет умеет делать это автоматически; он интерпретирует закодированные данные веб-формы, переданные через запросы типа GET или POST, и помещает их внутрь параметра `request`. Доступ к ним можно получить с помощью метода `getParameter()`.

GET, POST и «дополнительный путь»

Существует два распространенных способа передачи информации от веб-браузера к сервлету или CGI-программе. Обычно для этого применяется запрос типа POST: клиент кодирует информацию и отправляет ее через поток в вашу программу, где будет выполнено дальнейшее декодирование. Этот способ подходит для передачи через веб-формы (или напрямую) больших объемов данных, в том числе и файлов. Второй способ подразумевает кодирование информации внутри URL-адреса, по которому выполняется запрос. Чаще всего это делается с помощью

запроса типа GET. В этом случае веб-браузер кодирует параметры и добавляет их в конец строки с URL-адресом. На стороне сервера они будут декодированы и переданы в приложение.

Как мы уже говорили в главе 14, при запросе типа GET параметры добавляются в конец URL-адреса в виде пар «ключ/значение», разделенных амперсандом (символом «&»); перед первым параметром указывается вопросительный знак («?»). После этого любые спецсимволы в адресе (такие как пробелы, «?» или «&») должны быть закодированы.

Данные также можно передавать через так называемый «*дополнительный путь*». Определив местонахождение вашего сервлета (или CGI-приложения), сервер отправит ему оставшуюся часть URL-адреса. Давайте рассмотрим следующие адреса:

```
http://www.myserver.example/servlets/MyServlet
http://www.myserver.example/servlets/MyServlet/foo/bar
```

Допустим, сервер ассоциирует первый URL-адрес с сервлетом под названием `MyServlet`. При переходе по второму адресу происходит все то же самое, но при этом фрагмент «*/foo/bar*» рассматривается как «дополнительный путь» и становится доступным через метод `getExtraPath()` из параметра `request`. Такой подход помогает делать URL-адреса более логичными и понятными для человека — особенно это касается данных, состоящих из отдельных документов.

В HTML-формах на клиентской стороне можно использовать оба типа запросов (и POST, и GET); для этого в атрибуте `action` тега `form` необходимо указать одно из значений — «`post`» или «`get`». Кодированием данных занимается веб-браузер, а декодированием — сервлет.

При передаче содержимого формы на сервер клиент использует тип «`application/x-www-form-urlencoded`». Контейнер автоматически разбирает такие данные, предоставляя к ним доступ с помощью метода `getParameter()`. Но вы также можете получить их напрямую в необработанном виде, если прочитаете входящий поток.

Какой тип запросов лучше использовать — GET или POST?

Для пользователя главная разница между запросами типа GET и POST заключается в том, что GET хранит всю закодированную информацию в URL-адресе, который выводится прямо в веб-браузере. Этот адрес (например, результат поиска) можно скопировать и поделиться

им со своим другом по электронной почте, или сохранить в закладках на будущее. Информация, передаваемая в запросе типа POST, не видна пользователю и исчезает сразу после отправки на сервер. Такая разница в поведении не случайна, ведь запросы типа GET и POST даже на уровне протокола имеют разное предназначение. Результат выполнения операции GET по определению не должен быть связан с действиями, которые приводят к далеко идущим последствиям (например, к покупке товаров, находящихся в виртуальной корзине интернет-магазина). Именно для этого и был придуман запрос типа POST. Вот почему веб-браузеры предупреждают нас о повторной передаче данных, когда мы перезагружаем страницу с результатами отправки формы.

Использование дополнительного пути может пригодиться в сервлетах, которые принимают файлы или обрабатывают целый диапазон URL-адресов, обладающих легкой для восприятия структурой. Такие адреса более заметны, их легче запомнить.

Сервлет `ShowParameters`

Этот пример мы сделали не таким скучным, как предыдущий. Его главная задача — выводить значения всех переданных ему параметров. Сначала обеспечим обработку запросов типа GET, а затем добавим поддержку операций POST, внеся некоторые простые изменения. Вот код сервлета:

```
import java.io.*;
import javax.servlet.http.*;
import java.util.*;
public class ShowParameters extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse
response)
      throws IOException
   {
      showRequestParameters( request, response );
   }
   void showRequestParameters(HttpServletRequest request,
      HttpServletResponse response)
      throws IOException
   {
      response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
        out.println(
            "<html><head><title>Список параметров</title></head><body>"
            + "<h1>Параметры</h1><ul>");
        Map<String, String[]> params = request.getParameterMap();
        for ( String name : params.keySet() )
        {
            String [] values = params.get( name );
            out.println("<li>"+ name +" = "+ Arrays.asList(values) );
        }
        out.close( );
    }
}
```

Здесь, как и в предыдущем примере, переопределяется метод `doGet()`. Запросы, которые в него поступают, мы перенаправляем в созданный нами метод под названием `showRequestParameters()`; там мы перебираем ассоциативный массив с именами и значениями параметров, полученный из запроса с помощью метода `getParameterMap()`, и выводим его содержимое. Обратите внимание: если в клиентской веб-форме параметр указан несколько раз, он будет иметь больше одного значения, и тогда ассоциативный массив будет содержать объект `String[]`. Чтобы результат выглядел аккуратней, мы поместили каждый параметр внутрь HTML-тега `<li>`.

Как видно, наш сервлет будет реагировать на любой URL-адрес, содержащий запрос типа GET. Давайте добавим в наш вывод собственную веб-форму для отправки данных командой POST. Чтобы обрабатывать такие запросы, переопределим метод `doPost()`. Мы могли бы просто перенаправлять вызовы в метод `showRequestParameters()`, но можно сделать еще проще. API-интерфейс позволяет делать запросы типа GET и POST взаимозаменяемыми, так как декодированием параметров все равно занимается контейнер сервлета. Поэтому мы можем просто делегировать операцию `doPost()` методу `doGet()`.

Добавьте в код примера следующий метод:

```
public void doPost( HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException
{
    doGet( request, response );
}
```

Теперь давайте добавим в наш вывод веб-форму, с помощью которой пользователь сможет указывать параметры и передавать их на сервер. Добавьте следующий код в метод `showRequestParameters()`, прямо перед вызовом `out.close()`:

```
out.println("</ul><p><form method=\"POST\" action=\""
        + request.getRequestURI() + "\">"
   + "Поле 1 <input name=\"Field 1\" size=20><br>"
   + "Поле 2 <input name=\"Field 2\" size=20><br>"
   + "<br><input type=\"submit\" value=\"Отправить\"></form>"
);
```

В качестве значения для атрибута `action` в веб-форме выступает URL-адрес нашего сервлета, который и получит введенные данные. Для определения местоположения сервлета используется метод `getRequestURI()`. Мы указали для атрибута `method` операцию POST, но вы можете заменить ее на GET, чтобы проверить в деле оба типа запросов.

Пока что вы не увидели ничего особо выдающегося, но это только начало. В следующем примере мы научимся хранить клиентские данные между запросами посредством пользовательских сессий. Но прежде чем идти дальше, стоит упомянуть об одном полезном сервлете под названием `SnoopServlet`, который входит в стандартную поставку и по своим функциям очень похож на наш предыдущий пример.

Управление пользовательскими сессиями

Одной из самых полезных особенностей технологии Servlet API является механизм управления пользовательскими сессиями. Он позволяет сервлету хранить информацию между несколькими запросами, возникающими в результате загрузки пользователем разных страниц; этот процесс еще называют «поддержанием состояния». Связывание разных веб-страниц может пригодиться для решения разных задач, таких как обеспечение входа в систему или отслеживание покупки товаров в Интернет-магазине. Сессии в чем-то похожи на поля сервлета — они позволяют сохранять данные между вызовами метода `service()` и его производных.

Учет сессий выполняется на уровне контейнера сервлета; в большинстве случаев вам не придется вникать в подробности этого процесса. Поддержка сессий может достигаться либо за счет *файлов cookie на клиентской стороне*, либо посредством *переопределения URL-адресов*. Первый способ является частью стандартного для протокола HTTP

механизма, который позволяет переложить хранение информации о состоянии на веб-браузер. Cookie — это всего лишь атрибут вида «ключ/значение», который создается на серверной стороне, хранится на клиенте и возвращается обратно каждый раз, когда происходит доступ к определенному диапазону URL-адресов на указанном веб-сервере. С помощью сессий можно отслеживать активность как одного, так и множества пользователей.

Второй подход подразумевает добавление информации для отслеживания сессии непосредственно в URL-адрес; для этого могут использоваться закодированные параметры или дополнительный путь. Мы используем термин «переопределение», потому что перед появлением на клиентской стороне URL-адрес переопределяется сервером, а дополнительная информация из него опять передается сервлету. Для поддержки этого механизма сервлет должен кодировать все URL-адреса (например, в ссылках), которые появляются на выводимых им страницах, используя специальный метод из объекта `HttpServletResponse` (об этом мы поговорим позже). Если вы хотите, чтобы ваше приложение могло взаимодействовать с веб-браузерами, в которых файлы cookie не поддерживаются или просто выключены, вам придется разрешить переопределение URL-адресов на сервере. Хотя многие веб-сайты просто отказываются работать без доступа к файлам cookie.

Для программиста, который занимается написанием сервлета, информация о состоянии доступна в виде экземпляра `HttpSession`; в сущности, это хеш-таблица, способная хранить любые объекты, которые вы хотите сделать частью сессии. Сами объекты остаются на серверной стороне, а клиент получает специальный идентификатор (посредством файлов cookie или переопределения URL-адреса). При запросе этот идентификатор привязывается к сессии, а сессия, в свою очередь, ассоциируется с сервлетом.

Сервлет `ShowSession`

Сервлет, представленный ниже, показывает пример хранения некой строковой информации для отслеживания сессии:

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import java.util.Enumeration;
```

```
public class ShowSession extends HttpServlet {
   public void doPost(
      HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
   {
      doGet( request, response );
   }
   public void doGet(
      HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
   {
      HttpSession session = request.getSession();
boolean clear = request.getParameter("clear") != null;
if ( clear )
   session.invalidate();
else {
   String name = request.getParameter("Name");
   String value = request.getParameter("Value");
   if ( name != null && value != null )
      session.setAttribute( name, value );
}
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println(
   "<html><head><title>Содержимое сессии</title></head><body>");
if ( clear )
   out.println("<h1>Сессия очищена:</h1>");
else {
   out.println("<h1>В этой сессии:</h1><ul>");
   Enumeration names = session.getAttributeNames();
   while ( names.hasMoreElements() ) {
      String name = (String)names.nextElement();
      out.println( "<li>"+name+" = " +session.getAttribute(
         name ) );
   }
}
out.println(
   "</ul><p><hr><h1>Add String</h1>"
   + "<form method=\"POST\" action=\""
   + request.getRequestURI() +"\">"
   + "Ключ: <input name=\"Name\" size=20><br>"
```

```
            + "Значение: <input name=\"Value\" size=20><br>"
            + "<br><input type=\"submit\" value=\"Отправить\">"
            + "<input type=\"submit\" name=\"clear\" value=\"Очистить\"></
form>"
        );
    }
}
```

Загрузив этот сервлет в веб-браузере, вы увидите форму с возможностью ввода ключа и значения. Строковое значение сохраняется в объекте сессии по заданному ключу. При каждом запросе сервлет выводит список всех элементов, связанных с сессией. При добавлении новых значений содержимое сессии будет увеличиваться. В этом примере сессия очищается только после перезапуска веб-браузера или сервера.

Принципиально этот пример мало чем отличается от сервлета `ShowParameters`. Метод `doGet()` генерирует форму, которая шлет данные обратно на сервлет с помощью операции POST. Чтобы обрабатывать запросы любых типов, мы делегируем вызовы `doPost()` методу `doGet()`. При обработке запроса мы первым делом пытаемся получить пользовательскую сессию из параметра `request`, используя метод `getSession()`. Объект `HttpSession` имеет вид хеш-таблицы. Он содержит методы `setAttribute()` и `getAttribute()`, первый из которых принимает строковое имя и объект типа `Object`. В нашем примере мы также используем метод `getAttributeNames()`, чтобы перечислить и вывести все значения, которые хранятся внутри сессии в момент вызова.

По умолчанию, если текущая сессия отсутствует, метод `getSession()` создает новую. Для проверки наличия сессии или ручного управления процессом ее создания предусмотрена перегруженная версия метода `getSession(false)`, которая ничего сама не создает, и при остутствии сессии возвращает `null`. Как вариант с помощью метода `isNew()` вы можете узнать, является ли сессия новой (то есть использовалась ли она ранее). Чтобы немедленно очистить сессию, используется метод `invalidate()`. После его вызова мы больше не можем получить доступ к объекту `HttpSession`, поэтому в нашем примере мы выводим сообщение «Сессия очищена». Сессии сами могут становиться недоступными по истечении какого-то периода. Продолжительность этого периода можно регулировать с помощью сервера приложений или в файле *web.xml* (параметр «session-timeout» и разделе «session config»). Существует интерфейс, с помощью которого можно узнать, когда истекает время работы сессии, но об этом мы поговорим чуть позже. С точки зрения само-

го приложения сессия либо просто отсутствует, либо будет создана при следующем запросе. Пользовательские сессии соблюдают строгую принадлежность к конкретному веб-приложению и не могут передаваться за его пределы.

Ранее мы упоминали, что для поддержки веб-браузеров, в которых выключены файлы cookie, необходимо использовать переопределение URL-адресов. Для этого любые адреса, которые мы генерируем в рамках веб-страницы, должны быть пропущены через метод `encodeURL()` из объекта `HttpServletResponse`. Этот метод принимает строку URL и модифицирует ее только в том случае, если переопределение действительно необходимо. Обычно, если файлы cookie доступны, он возвращает строку без всяких изменений. В нашем предыдущем примере мы могли закодировать URL-адрес формы, прежде чем передавать его на клиентскую сторону — это обеспечило бы нам поддержку любых браузеров.

Сервлет `ShoppingCart`

Взяв за основу предыдущий пример, мы создадим сервлет под названием `ShoppingCart`, который можно использовать как часть интернет-магазина. С его помощью пользователи, прежде чем совершить покупку, смогут выбирать товары и добавлять их в свою виртуальную корзину. Сгенерированная нами страница будет выглядеть не очень эстетично, но вы можете это исправить, попросив своего знакомого веб-дизайнера навести марафет с помощью CSS-файла (смайлик). Нас же в первую очередь интересует работа с технологией Servlet API:

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import java.util.Enumeration;
public class ShoppingCart extends HttpServlet
{
   String [] items = new String [] {
      "Сверчки в шоколаде", "Малиновые тараканы",
      "Мухи с маслом", "Пастилки со вкусом цыпленка(tm)" };
   public void doPost(
      HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException
   {
      doGet( request, response );
```

```
}
public void doGet(
   HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException
{
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   // получаем или создаем информацию о сессии
   HttpSession session = request.getSession();
   int [] purchases = (int [])session.getAttribute("purchases");
   if ( purchases == null ) {
      purchases = new int [ items.length ];
      session.setAttribute( "purchases", purchases );
   }
   out.println( "<html><head><title>Корзина для покупок</title>"
         + "</title></head><body><p>" );
   if ( request.getParameter("checkout") != null )
      out.println("<h1>Благодарим за покупку!</h1>");
   else {
      if ( request.getParameter("add") != null ) {
         addPurchases( request, purchases );
         out.println(
            "<h1>Товар добавлен. Продолжайте, пожалуйста</h1>");
      } else {
         if ( request.getParameter("clear") != null )
            for (int i=0; i<purchases.length; i++)
               purchases[i] = 0;
         out.println("<h1>Пожалуйста, выберите понравившийся товар!</h1>");
         }
         doForm( out, request.getRequestURI() );
      }
      showPurchases( out, purchases );
      out.close();
   }
   void addPurchases( HttpServletRequest request, int [] purchases ) {
      for (int i=0; i<items.length; i++) {
         String added = request.getParameter( items[i] );
         if ( added !=null && !added.equals("") )
            purchases[i] += Integer.parseInt( added );
      }
   }
```

```
void doForm( PrintWriter out, String requestURI ) {
   out.println( "<form method=POST action="+ requestURI +">" );
   for(int i=0; i< items.length; i++)
      out.println( "Quantity <input name=\"" + items[i]
         + "\" value=0 size=3> of: " + items[i] + "<br>");
   out.println(
      "<p><input type=submit name=add value=\"Добавить в корзину\">"
      + "<input type=submit name=clear value=\"Очистить корзину\">"
      + "<input type=submit name=checkout value=\"Сделать покупку\">"
      + "</form>" );
}
void showPurchases( PrintWriter out, int [] purchases )
      throws IOException {
      out.println("<hr><h2>Ваша корзина для покупок</h2>");
      for (int i=0; i<items.length; i++)
         if ( purchases[i] != 0 )
            out.println( purchases[i] +" "+ items[i] +"<br>" );
   }
}
```

Стоит отметить, что сервлет `ShoppingCart` хранит некоторые данные в полях экземпляра, а именно строковой массив со списком товаров. Мы исходим из того, что всем покупателям предлагается одинаковый ассортимент. В противном случае нам бы пришлось генерировать список продуктов на лету или помещать его внутрь пользовательской сессии. Мы не можем хранить в полях экземпляра сервлета данные, которые относятся к конкретному запросу или пользователю.

Вызовы `doPost()` делегируются методу `doGet()`, который генерирует основную часть страницы и форму для сбора новых данных — такой подход мы уже видели в предыдущем примере. Весь процесс мы разбили на несколько вспомогательных методов: `doForm()`, `addPurchases()` и `showPurchases()`. Наша форма содержит три кнопки: одна для добавления товаров в корзину, другая для ее очистки и еще одна для осуществления покупки. Что бы вы ни выбрали, сервлет выведет содержимое корзины. В зависимости от нажатой кнопки (о чем можно судить по имени параметра) мы выполняем одно из трех действий: добавляем новые товары, очищаем список или благодарим за покупку.

В форме, сгенерированной методом `doForm()`, используется список товаров, выставленных на продажу. Как и предыдущих примерах, атрибуту `action` присваивается адрес нашего сервлета. После этого

целочисленный массив под названием purchases помещается в пользовательскую сессию. Он хранит количество единиц товара, которые пользователь хочет купить, и создается на основе данных, полученных из сессии. В новой сессии массива еще нет, поэтому при попытке его получения метод getAttribute() вернет **null**; в таком случае нам придется создать пустой массив и заполнить его данными. В форме используются названия из массива items, поэтому при совершении покупки мы можем их проверять с помощью метода getParameter(), увеличивая значение соответствующего элемента массива purchases на количество запрошенных товаров. Нам также необходимо выполнять проверку на пустые строки, которые в некоторых веб-браузерах обозначают неиспользованные поля. В конце метод showPurchases() выводит названия и количество купленных товаров, перебирая массив purchases.

Файлы cookie

В нашем предыдущем примере сессии могут существовать до тех пор, пока вы не закроете веб-браузер или не перезапустите сервер. Но при ручном использовании файлов cookie можно создать более долгосрочный и устойчивый механизм для отслеживания и идентификации пользователей. Чтобы отправить файл cookie на клиентскую сторону, можно создать объект javax.servlet.http.Cookie и добавить его в ответ сервлета с помощью метода addCookie(). Позже клиент сможет извлечь этот файл и найти с его помощью какую-то информацию в базе данных. Следующий сервлет передает вашему веб-браузеру файл cookie с ключом «Learningjava», отображая его при повторной загрузке страницы:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieCutter extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse
response)
      throws IOException, ServletException
   {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter( );
      if ( request.getParameter("setcookie") != null ) {
         Cookie cookie = new Cookie("Learningjava", "Куки!");
         cookie.setMaxAge(3600);
```

```
        response.addCookie(cookie);
        out.println("<html><body><h1>Файл cookie установлен...</h1>");
    } else {
        out.println("<html><body>");
        Cookie[] cookies = request.getCookies( );
        if ( cookies.length == 0 ) {
            out.println("<h1>Не удалось найти файлы cookie...</h1>");
        } else {
            for (int i = 0; i < cookies.length; i++)
                out.print("<h1>Ключ: "+ cookies[i].getName() + "<br>"
                    + "Значение: " + cookies[i].getValue() + "</h1>" );
        }
            out.println("<p><a href=\""+ request.getRequestURI()
                +"?setcookie=true\">"
                +"Сбросить файл cookie.</a>");
        }
        out.println("</body></html>");
    }
}
```

Этот код просто выводит ключи и значения всех файлов cookie, полученных из запроса с помощью метода `getCookies()`. Мы предоставили ссылку в стиле команды GET, которая указывает на наш сервлет; она содержит параметр `setcookie`, сигнализирующий о том, что мы должны установить файл cookie. В этом случае мы создаем объект `Cookie`, используя заданные ключ и значение и добавляем его в ответ с помощью метода `addCookie()`. Мы сделали так, что максимальный срок годности файла cookie равен 3600 секундам — то есть через час он будет удален из браузера (о том, как отслеживать файлы cookie в разных сессиях, мы поговорим позже). Если указать отрицательное значени файл cookie будет считаться временным и исчезнет вместе с закрытием браузера. Чтобы немедленно удалить любой существующий файл cookie, достаточно указать для него срок годности, равный нолю.

У объекта `Cookie` есть еще два интересных метода: `setDomain()` и `setPath()`. Они позволяют указать доменное имя и путь, по которому клиент должен передавать файл cookie. Если вы создаете апплет для выполнения покупок на сайте **Amazon.com**, вам уж точно не понравится, если ваш файл cookie будет отправлен на **eBay.com**. Хотя в реальности этого просто не может произойти, так как по умолчанию используется домен сервера, который отправляет файл cookie (из соображений безопасности любой другой домен будет игнорироваться). Изначально путь

равен базовому URL-адресу сервлета, но вы можете вручную указать более широкий (или узкий) диапазон адресов, не выходя за пределы текущего домена.

Интерфейс `ServletContext`

Чтобы получить доступ к окружению сервера, веб-приложения используют интерфейс `ServletContext`, ссылку на который можно извлечь с помощью метода `getServletContext()` из объекта `HttpServlet`:

```
ServletContext context = getServletContext();
```

Каждое веб-приложение имеет собственный экземпляр `ServletContext`. Этот контекст предоставляет некое общее пространство, в рамках которого сервлеты могут взаимодействовать между собой и обмениваться объектами. Чтобы поместить объект в контекст, используйте метод `setAttribute()`; для получения объекта по его имени предусмотрен метод `getAttribute()`:

```
context.setAttribute("myapp.statistics", myObject);
Object stats = context.getAttribute("myapp.statistics");
```

Названия атрибутов, начинающиеся с «`java.`» и «`javax.`», зарезервированы платформой Java. Чтобы избежать конфликтов, лучше придерживаться общепринятого соглашения об именовании пакетов.

Интерфейс `ServletContext` поддерживает отслеживание событий; с его помощью вы сможете добавлять в контекст сервлета новые элементы при запуске сервера, и удалять их при завершении работы. Это хороший инструмент для создания разделяемых сервисов, пример которых вы увидите в следующем разделе, где речь пойдет об асинхронных сервлетах.

У контекста сервлета есть стандартный атрибут, с помощью которого можно получить ссылку на локальную рабочую директорию, представленную объектом `java.io.File`. Эта директория является уникальной для каждого отдельного веб-приложения. Однако несмотря на свой «временный» статус она не очищается даже после завершения работы, поэтому, если вы хотите создавать внутри нее какие-либо файлы, вам нужно использовать для этого специальные методы (если только вы не хотите, чтобы ее содержимое сохранялось между запусками сервера). Например:

```
File tmpDir = (File)context.getAttribute("javax.servlet.context.tempdir");
File tmpFile = File.createTempFile( "appprefix", "appsuffix", tmpDir );
```

Интерфейс `ServletContext` также предоставляет прямой доступ к файлам веб-приложения, начиная с его корневой директории. Метод `getResource()` похож на свой одноименный аналог из класса `Class` (см. главу 12). В качестве аргумента он принимает путь, возвращая специальный локальный URL-адрес для доступа к соответствующему ресурсу. В нашем случае используется путь к базовой директории сервлета (внутри WAR-архива). С помощью этого метода можно получать ссылки на файлы, включая те, что находятся в директории *WEB-INF*. К примеру, вот как сервлет будет извлекать входящий поток для собственного файла *web.xml*:

```
InputStream in = context.getResourceAsStream("/WEB-INF/web.xml");
```

Мы также можем получить URL-адрес изображения, принадлежащего сервлету:

```
URL bunnyURL = context.getResource("/images/happybunny.gif");
```

С помощью метода `getResourcePaths()` можно получить список всех ресурсных файлов, доступных по заданному пути. В качестве результата выступает коллекция `java.util.Set`, которая содержит строковые имена ресурсов. Например, для пути / перечисляются все файлы в корне архива WAR; указав */WEB-INF/*, вы получите как минимум файл *web.xml* и директорию *classes*.

Интерфейс `ServletContext` также является фабрикой объектов `RequestDispatcher`; мы не будем рассматривать их в этой книге, но вы должны знать, что они позволяют перенаправлять запросы другим сервлетам или включать их результаты в свой собственный ответ.

Асинхронные сервлеты

Это не совсем простая тема, но, чтобы завершить рассмотрение технологии Servlet API, мы обсудим ее сейчас. Сервлеты могут работать в асинхронном режиме, позволяя методу `service()` завершаться мгновенно, но удерживая соединение с клиентом до тех пор, пока результат не будет готов. В этом состоянии сервлет не потребляет много ресурсов и не блокирует поток выполнения своего контейнера. Данный подход предназначен для поддержки неблокирующих сервисов, написанных в стиле технологии NIO, которую мы обсуждали в главах 13 и 14.

Асинхронные сервлеты прекрасно подходят для обработки медленных процессов, учитывая возможность выполнять эффективную выборку или получать по-настоящему асинхронные уведомления о заверше-

нии работы. Во время обсуждения пакета NIO мы говорили, что одним из ограничений при масштабировании веб-сервисов является ресурсоемкость потоков выполнения. Потокам требуется много вычислительных мощностей, поэтому нельзя просто позволять им блокировать работу и ждать завершения задачи — это будет неэффективно. Как вы уже знаете, пакет NIO поддерживает стиль программирования, при котором один поток способен обслуживать большое количество сетевых соединений. Этот же подход можно использовать и в асинхронных сервлетах. Идея заключается в том, что задача выполняется в фоновом режиме, а запрос сервлета тем временем откладывается. Если спроектировать фоновый обработчик таким образом, чтобы он мог управлять задачами без задержек и ожидания (с использованием выборок или асинхронных уведомлений), у потока выполнения не будет причин блокироваться.

Позже в этой главе мы напишем простой тестовый сервлет под названием `WaitServlet`, который, прежде чем вернуть результат, ждет в течение указанного времени. Это ярчайший пример *неэффективного* использования потоков. Он блокирует поток выполнения (уходя в спящий режим) до тех пор, пока не придет время завершать транзакцию. В следующем примере мы забежим немного вперед и создадим более эффективную версию этого компонента, `BackgroundWaitServlet`, которая во время ожидания не блокирует ни одного потока в контейнере сервлета.

Прежде чем начать, давайте проверим условия, при которых от асинхронного сервлета будет какая-то польза: есть ли у нас эффективный способ узнавать состояние выполнения задач или получать уведомления об их завершении, не блокируя поток? (Важно подумать об этом заранее, чтобы все не свелось к простому перемещению блокируемых потоков из сервлета в какой-то другой участок веб-приложения.) Ответ положительный: в нашем случае мы можем использовать таймер, который будет уведомлять нас об истечении времени. Эффективная реализация таймера, такая как `java.util.Timer`, способна обрабатывать множество запланированных запросов в одном единственном потоке. Мы также воспользуемся интерфейсом `ScheduledExecutorService` из пакета `java.util.concurrent`. Он будет работать в фоне и вызывать задачи `Runnable` с заданной задержкой, играя роль идеального разделяемого сервиса для нашего асинхронного сервлета.

Следующий код возвращает простейшую страницу с пятисекундной задержкой. Разницу между этим и любым другим сервлетом, приведенным в этой главе, можно будет увидеть невооруженным глазом, отправив серверу множество одновременных запросов. Асинхронная версия

будет ограничена только производительностью сетевого интерфейса операционной системы, а не количеством памяти на сервере, которая является куда более ценным ресурсом.

```
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.concurrent.*;
@WebServlet(
   urlPatterns={"/bgwait"},
   asyncSupported = true
)
public class BackgroundWaitServlet extends HttpServlet
{
public void doGet( HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
   {
      final AsyncContext asyncContext = request.startAsync();
      ScheduledExecutorService executor =
         (ScheduledExecutorService)request.getServletContext().
getAttribute(
         "BackgroundWaitExecutor");
      executor.schedule( new RespondLaterJob( asyncContext ), 5,
         TimeUnit.SECONDS );
   }
}
class RespondLaterJob implements Runnable
{
   private AsyncContext asyncContext;
   RespondLaterJob( AsyncContext asyncContext ) {
      this.asyncContext = asyncContext;
   }
   @Override
   public void run()
   {
      try {
         ServletResponse response = asyncContext.getResponse();
         response.setContentType("text/html");
         PrintWriter out = response.getWriter();
         out.println(
```

```
                "<html><body><h1>Ответ сервлета WaitServlet</h1></body></html>"
            );
        } catch ( IOException e ) { throw new RuntimeException( e ); }
        asyncContext.complete();
    }
}
```

Мы добавили в этот пример аннотацию `WebServlet`, чтобы показать атрибут `asyncSupported`. Этот атрибут должен быть установлен для всех сервлетов и их фильтров (их мы обсудим позже), которые участвуют в обработке запроса.

Наша реализация метода `doGet()` выглядит довольно просто. Первым делом инициализируется асинхронное поведение. Для этого из параметра `request` вызывается метод `startAsync()`, который возвращает объект `AsyncContext`. Этот объект представляет контекст вызывающей стороны и вместе с запросом к сервлету включает в себя ответ, который тот должен вернуть. На этом этапе мы можем выбрать любые средства для обслуживания запроса; единственное условие: наша задача должна иметь доступ к объекту `AsyncContext`, чтобы позже с его помощью отправить результат и завершить транзакцию.

В вышеприведенном примере экземпляру `ScheduledExcecutorService`, который был получен по имени («`BackgroundWaitExecutor`») из контекста сервлета, передается наша версия объекта `Runnable` (о том, как сервис оказался в контексте сервлета, мы поговорим чуть позже). Мы создали задачу `RespondLaterJob`, которая реализует интерфейс `Runnable`, и привязали к ней объект `AsyncContext` для дальнейшего использования. При выполнении задачи мы получим ответ сервлета из контекста и отправляем его, как обычно. В самом конце из объекта `AsyncContext` нужно будет вызвать метод `complete()`, чтобы завершить работу и вернуть ответ клиенту.

Последний шаг имеет несколько любопытных моментов. Во-первых, нам не обязательно вызывать метод `complete()` сразу после выполнения записи в объект ответа. Вместо этого мы могли бы записать часть результата и вернуться в спящий режим, ожидая, когда наш сервис нас «разбудит» и предоставит новые данные. Именно так и следует работать с источником информации из пакета NIO. Во-вторых, вместо того чтобы вызывать метод `complete()` для окончательного оформления результата, мы могли бы воспользоваться альтернативным методом `dispatch()`, который позволяет перенаправлять запрос другому сервлету (таким образом можно построить целую цепочку). Следующий сервлет мог бы записать

дополнительные данные или просто обработать запрос с помощью ресурсов, помещенных в контекст предыдущим сервлетом. Метод `dispatch()` принимает URL-адрес, ведущий к нужному сервлету; если не указать ни одного аргумента, запрос будет передан изначальному обработчику.

Хорошо, так как же наш экземпляр `ScheduledExecutorService` попал в контекст сервлета? Лучшим средством для отслеживания разделяемых сервисов и ресурсов внутри контекста сервлета является объект `ServletContextListener`. Он содержит два метода для управления жизненным циклом, с помощью которых при запуске и завершении работы сервлета можно подключать и отключать различные сервисы. Мы можем установить этот объект стандартным способом: поместив его в WAR-архив и пометив его класс аннотацией `WebListener`.

```
import javax.servlet.*;
import javax.servlet.annotation.*;
import java.util.concurrent.*;
@WebListener
public class BackgroundWaitService implements ServletContextListener
{
   ScheduledExecutorService executor;
   public void contextInitialized( ServletContextEvent sce )
   {
      this.executor = Executors.newScheduledThreadPool( 3 );
      sce.getServletContext().setAttribute( "BackgroundWaitExecutor",
         executor );
   }
   public void contextDestroyed(ServletContextEvent sce)
   {
      ScheduledExecutorService executor =
         Executors.newScheduledThreadPool( 3 );
      executor.shutdownNow();
   }
}
```

WAR-файлы и развертывание веб-приложений

Как уже говорилось во введении к этой главе, WAR-файл — это архив, который содержит все компоненты веб-приложения: class-файлы для сервлетов и веб-сервисов, JSP-сценарии, HTML-страницы, изображения и дру-

гие ресурсы. Это просто особая версия JAR-файла (который, в свою очередь, является разновидностью ZIP-архива) с директориями, отведенными для Java-кода, и одним конфигурационным файлом под названием *web.xml*, благодаря которому сервер приложений знает, что и как нужно запускать. Все WAR-файлы имеют расширение *.war*, но создавать и просматривать их можно с помощью стандартной утилиты *jar*. Воспользовавшись этой утилитой, вы можете сами увидеть, как выглядит типичный WAR-архив:

```
$ jar tvf shoppingcart.war
   index.html
   purchase.html
   receipt.html
   images/happybunny.gif
   WEB-INF/web.xml
   WEB-INF/classes/com/mycompany/PurchaseServlet.class
   WEB-INF/classes/com/mycompany/ReturnServlet.class
   WEB-INF/lib/thirdparty.jar
```

По умолчанию в процессе развертывания имя WAR-файла используется в качестве корневого пути для веб-приложения — в нашем случае это *shoppingcart*. Таким образом, если установить это веб-приложение на сервер **http://www.example.ru**, его URL-адрес будет выглядеть как **http://www.example.ru/shoppingcart/**, и все ссылки на его документы, изображения и сервлеты будут начинаться с этого пути. Самый верхний уровень WAR-архива становится корневой (базовой) директорией для хранения файлов. Файл *index.html* доступен по базовому URL-адресу, о котором мы только что упоминали, а ссылка на изображение *happybunny.gif* выглядит как **http://www.example.ru/shoppingcart/images/happybunny.gif**.

WEB-INF (в верхнем регистре, через дефис) — это специальная директория, которая содержит код приложения и всю информацию о его развертывании. Она защищена веб-сервером, и ее содержимое нельзя увидеть на стороне клиента, даже если вы добавите ее название в базовый URL-адрес. Тем не менее содержащиеся в ней файлы можно загружать из контекста сервлета с помощью метода `getResource()` — это безопасное место для хранения программных ресурсов. В директории *WEB-INF* также находится файл *web.xml*, который мы еще обсудим в следующем разделе.

Директории *WEB-INF/classes* и *WEB-INF/lib* содержат, соответственно, *class*-файлы и библиотеки, упакованные в JAR-архивы. Директория

WEB-INF/classes автоматически добавляется в classpath, поэтому любые размещенные в ней классы (соблюдающие стандартный принцип структурирования пакетов) становятся доступными внутри веб-приложения. После этого в classpath добавляются JAR-файлы, размещенные в директории *WEB-INF/lib* (к сожалению, у нас нет возможности определить порядок их следования). Вы можете хранить свои классы в любом из этих каталогов. Хотя во время написания кода проще работать с более «свободной» директорией *classes*, используя каталог *lib* для вспомогательных классов и сторонних инструментов. JAR-файлы можно установить прямо в контейнер сервлета, чтобы они были доступны любому веб-приложению, запущенному на сервере. Так часто поступают с общими библиотеками, которые используются множеством разных программ. Однако место хранения библиотек не стандартизировано, и классы, установленные подобным образом, не будут автоматически перезагружаться при внесении в них изменений (об этой функции WAR-файлов мы поговорим позже). Технология Servlet API требует от сервера наличия специальной директории для этих дополнительных JAR-библиотек; к тому же их классы должны быть доступными для веб-приложения и использовать единый загрузчик.

Конфигурирование с помощью файла web.xml и аннотаций

web.xml — это конфигурационный файл в формате XML, в котором перечислены сервлеты и сопутствующие элементы, предназначенные для установки, относительные имена (URL-адреса), под которыми они будут устанавливаться, их начальные параметры и различные аспекты процесса развертывания, такие как безопасность и авторизация. Долгое время это был единственный способ сконфигурировать процесс установки веб-приложений на платформе Java. Но с выходом Servlet API 3.0 появились дополнительные варианты. Теперь большую часть этой работы можно проделать с помощью аннотаций. В своем первом примере, `HelloClient`, мы использовали аннотацию `WebServlet`, чтобы объявить сервлет и указать для него URL-адрес, по которому он будет доступен. Аннотации позволяют развернуть сервлет на сервере Tomcat без участия файла *web.xml*. С помощью технологии Servlet API 3.0 сервлет можно развернуть в императивном стиле, управляя процессом вручную, прямо в коде программы.

В этом разделе мы рассмотрим конфигурирование как в формате XML, так и с помощью аннотаций. Для большинства задач аннотации подходят лучше, но умение обращаться с файлом *web.xml* тоже может

пригодиться. Во-первых, файл *web.xml* позволяет переопределять аннотации, описанные в коде приложения. С его помощью вы можете менять конфигурацию на этапе развертывания, не компилируя заново свои классы. В целом этот подход имеет более высокий приоритет, чем аннотации, которые вообще могут быть проигнорированы сервером, если указать в файле *web.xml* атрибут под названием `metadata-complete`. Кроме того, может возникнуть необходимость в дополнительном конфигурировании (особенно это касается свойств контейнера сервлета), которое можно выполнить только в формате XML.

Мы будем исходить из того, что вы хотя бы поверхностно знакомы с технологией XML, но у вас всегда есть возможность просто скопировать приведенные нами примеры (подробности о работе с форматом XML на платформе Java можно узнать в главе 24). Давайте начнем с простого файла *web.xml* для нашего сервлета `HelloClient`. Он будет выглядеть следующим образом:

```
<web-app>
   <servlet>
      <servlet-name>helloclient1</servlet-name>
      <servlet-class>HelloClient</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>helloclient1</servlet-name>
      <url-pattern>/hello</url-pattern>
   </servlet-mapping>
</web-app>
```

На верхнем уровне документа находится элемент под названием `<web-app>`. Внутри него может быть указано множество разных записей, но основными являются теги `<servlet>` и `<servlet-mapping>`, предназначенные, соответственно, для объявления и развертывания сервлета. В теге `<servlet>` также можно указывать начальные и другие параметры; стоит отметить, что в этой секции должен содержаться только один экземпляр типа `Servlet`, хотя самих тегов `<servlet>` в файле *web.xml* может быть несколько.

Тег `<servlet>` должен содержать как минимум два информационных элемента: `<servlet-name>`, который идентифицирует сервлет в файле *web.xml*, и `<servlet-class>`, в котором указано имя класса сервлета. В этом примере мы использовали имя `helloclient1`, показывая тем самым, что при желании можно объявить и другие экземпляры одного и того же сервлета, предоставляя им отдельные начальные параметры

и т. д. Класс нашего сервлета называется `HelloClient`, хотя в настоящем приложении стоило бы указать полное имя пакета — например, `com.oreilly.servlets.HelloClient`.

Объявление может включать в себя один или несколько начальных параметров, доступ к которым можно получить в коде сервлета посредством метода `getInitParameter()` из объекта `ServletConfig`:

```
<servlet>
   <servlet-name>helloclient1</servlet-name>
   <servlet-class>HelloClient</servlet-class>
   <init-param>
      <param-name>foo</param-name>
      <param-value>bar</param-value>
   </init-param>
</servlet>
```

Теперь рассмотрим элемент `<servlet-mapping>`. Он предназначен для связывания экземпляра сервлета с URL-адресом на сервере:

```
<servlet-mapping>
   <servlet-name>helloclient1</servlet-name>
   <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

Здесь мы назначили нашему сервлету путь */hello* (при желании мы могли бы указать дополнительные теги `url-pattern`). Если позже мы назовем наш WAR-архив *learningjava.war* и развернем его на сервере **www.example.com**, то полный путь к сервлету будет иметь вид **http://www.example.com/hello**. Для каждого экземпляра сервлета (которых, напомним, может быть больше одного) можно указать несколько тегов `<servlet-mapping>`. К примеру, мы могли бы привязать один и тот же экземпляр к путям */hello* и */hola*. Тег `<url-pattern>` предоставляет очень гибкий механизм описания URL-адресов, но более подробно об этом вы узнаете в следующем разделе.

Напоследок нужно сказать, что, несмотря на совместимость вышеприведенного файла *web.xml* с некоторыми серверами приложений, формально наш пример является неполным, так как в нем отсутствует метаинформация о версиях формата XML и спецификации, которой этот файл отвечает. Для полного соответствия стандартам добавьте следующую строчку:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Стандарт Servlet API 2.5 требует, чтобы информация о версии файла *web.xml* описывалась на языке семейства XML Schema (о технологиях XML DTD и XML Schema мы поговорим в главе 24). В элемент `<web-app>` нужно вставить дополнительный код:

```
<web-app
   xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
   version="2.5">
```

Вполне возможно, что приложение будет работать, даже если оставить все как есть, но контейнеру сервлета будет труднее определять ошибки в вашей конфигурации и выдавать вам соответствующие сообщения.

Как мы уже знаем, эквивалентом этого XML-файла может служить однострочная аннотация:

```
@WebServlet(urlPatterns={"/hello", "/hola"})
public class HelloClient extends HttpServlet {
   ...
}
```

Здесь атрибут `urlPatterns` в аннотации `WebServlet` позволяет указывать сразу несколько шаблонов для URL-адресов; в файле *web.xml* для этого использовался тег `url-pattern`.

Привязка к URL-шаблонам

В предыдущем примере в качестве шаблона для тега `<url-pattern>` мы указали простую строку, `/hello`. С таким шаблоном наш сервлет можно будет вызвать только по базовому URL-адресу, в конце которого содержится путь */hello*. Но тег `<url-pattern>` поддерживает и более гибкие шаблоны, в том числе с использованием групповых символов.

Например, если указать строку `/hello*`, наш сервлет будет доступен по адресам **http://www.example.ru/learningjava/helloworld** или **.../hellobaby**. Вы даже можете разместить групповой символ перед расширением (например, `*.html` или `*.foo`), что позволит вызывать сервлет по любому пути, который заканчивается этими символами.

Благодаря использованию групповых символов можно добиться сразу нескольких совпадений. Представьте, что шаблоны заканчиваются на `/scooby*` и `/scoobydoo*`. Какой из них будет соответствовать URL-адресу **.../scoobydoobiedoo**? А что, если из-за группового символа перед расширением у нас будет третье потенциальное совпадение? Такие конфликтные ситуации решаются по определенным правилам.

В первую очередь берется любое точное совпадение. Например, адрес */hello* соответствует строке `/hello`, несмотря на наличие дополнительного шаблона `/hello*`. Если точного совпадения не найдено, контейнер ищет самое длинное префиксное соответствие. Таким образом, адрес */scoobydoobiedoo* удовлетворяет второму шаблону, `/scoobydoo*`, как более длинному и, вероятно, точному. Если и тут у нас нет подходящего варианта, контейнер переходит к групповым символам в суффиксах. На этом этапе окончание *.foo* соответствует шаблону `*.foo`. В случае отсутствия любых совпадений контейнер использует шаблон по умолчанию, `/*`, который подходит для любых адресов. Сервлет, связанный с шаблоном `/*`, отзывается на любой запрос, который не был перехвачен до этого момента. Если даже эта стандартная привязка отсутствует, запрос закончится неудачей, а клиент получит сообщение «404 Not Found».

Развертывание сервлета `HelloClient`

Закончив развертывание сервлета `HelloClient`, вы сможете легко добавлять в свой WAR-файл новые примеры, рассматриваемые в этой главе. В данном разделе вы научитесь ручному созданию WAR-файлов. Позже, в разделе «Создание WAR-файлов с помощью утилиты Ant» мы покажем вам более практичный подход к управлению приложениями с использованием этого популярного инструмента для сборки проектов. Полное собрание примеров из этой книги, упакованное в файл *learningjava.war*, можно найти по адресу **github.com/patniemeyer/learningjava**.

Прежде чем сгенерировать WAR-архив, необходимо создать директории *WEB-INF* и *WEB-INF/classes*. Если вы используете файл *web.xml*, поместите его внутрь *WEB-INF*. Класс *HelloClient.class* должен находиться в директории *WEB-INF/classes*. Чтобы создать файл *learningjava.war*, воспользуйтесь утилитой *jar* (в качестве «верхнего» уровня архива выберите директорию *WEB-INF*):

```
$ jar cvf learningjava.war WEB-INF
```

Вы также можете добавить в свой архив документы и другие ресурсы, указывая их имена после директории *WEB-INF*. Команда, приведенная выше, генерирует файл *learningjava.war*. Проверить его содержимое можно с помощью той же утилиты *jar*:

```
$ jar tvf learningjava.war
document1.html
WEB-INF/web.xml
WEB-INF/classes/HelloClient.class
```

Теперь осталось только найти для WAR-архива подходящее место на вашем сервере. Загрузите и установите приложение Apache Tomcat (если вы еще этого не сделали). Ваши WAR-файлы должны храниться в каталоге *webapps* внутри установочной директории сервера Tomcat. Поместите туда свой файл и запустите сервер. Если Tomcat использует стандартный номер порта, сервлет HelloClient должен быть доступен по одному из двух адресов: **http://localhost:8080/learningjava/hello** или **http://<yourserver>:8080/learningjava/hello**, где «<yourserver>» — доменное имя или IP-адрес вашего сервера. Если у вас возникли какие-то проблемы, ищите отчеты об ошибках в каталоге *logs* установочной директории Tomcat.

Перезагрузка веб-приложений

Все контейнеры сервлетов должны предоставлять средства для перезагрузки WAR-файлов; многие из них умеют перезагружать отдельные классы, которые подвергаются модификации. Эти функции являются частью спецификации сервлетов и могут быть особенно полезными в процессе разработки. Реализация перезагрузки веб-приложений может отличаться на разных серверах. Обычно достаточно предоставить новый WAR-файл вместо старого, поместив его в подходящую директорию (*webapps*, в случае с сервером Tomcat) — контейнер сам завершит работу текущей версии приложения и развернет вместо нее новую. В сервере Tomcat за автоматическую перезагрузку отвечает атрибут «autoDeploy» (включен по умолчанию), а на платформе WebLogic эта функция активируется вместе с режимом отладки.

Некоторые серверы, в том числе и Tomcat, распаковывают WAR-архивы в подкаталог директории *webapps*; в таких случаях вы сами можете указать корневую директорию (или «контекст») своего распакованного приложения в конфигурационном файле сервера. В таком режиме у вас может появиться возможность менять отдельные файлы,

что будет особенно удобно при редактировании HTML- и JSP-страниц. Сервер Tomcat автоматически перезагружает архив с веб-приложением (если вы только не запретили ему это делать), поэтому вам достаточно просто скопировать новый WAR-файл вместо старого, и он, при необходимости, будет заново развернут. В некоторых случаях требуется перезапустить сам сервер, чтобы все изменения вступили в силу. Универсальный совет: если сомневаетесь — выключите и включите снова.

Сервер Tomcat также предоставляет пакет для клиентской стороны, который интегрируется с утилитой Ant, автоматизируя развертывание (в том числе и повторное) и сборку приложений. Программу Ant мы обсудим позже в этой главе.

Вывод ошибок и индексные страницы

Одним из главных признаков профессионально написанного приложения является надлежащая обработка ошибок. Ничто так не раздражает пользователя, как причудливые страницы с непонятной технической информацией, особенно если вместо них он ожидал увидеть рецепт какого-нибудь новогоднего блюда. В файле *web.xml* можно указать документы или сервлеты, которые будут обрабатывать ошибки, возникающие в различных ситуациях, а также специальные *индексные файлы*, которые отображаются, если путь ведет к директории. На сегодняшний день этот аспект веб-приложения нельзя сконфигурировать с помощью аннотаций.

С помощью одного или нескольких тегов `<error-page>` можно объявлять страницы или сервлеты, которые будут отвечать за обработку различных ошибок с разными кодами, такими как «404 Not Found» и «403 Forbidden»:

```
<web-app>
...
   <error-page>
      <error-code>404</error-code>
      <location>/notfound.html</location>
   </error-page>
   <error-page>
      <error-code>403</error-code>
      <location>/secret.html</location>
   </error-page>
```

У вас есть возможность описывать ошибки не только в виде кодов, но и с помощью типов исключений, которые могут быть сгенерированы сервлетом. Например:

```
<error-page>
   <exception-type>java.lang.IOException</exception-type>
   <location>/ioexception.html</location>
</error-page>
```

Этот код перехватывает исключения типа `IOException`, сгенерированные сервлетом в рамках веб-приложения, и отображает страницу *ioexception.html*. Если будет сгенерировано исключение `ServletException` (или его производная), и в блоках `<error-page>` не найдется подходящего обработчика, контейнер попытается извлечь из экземпляра `ServletException` вложенное исключение (которое вызвало ошибку) и сопоставить его с объявленными страницами. Спецификация Servlet API 3.0 позволяет назначить страницу, которая будет перехватывать ошибки с любыми кодами и типами исключений:

```
<error-page>
   <location>/anyerror.html</location>
</error-page>
```

Как мы уже упоминали, для обработки ошибок вместо статического документа можно использовать сервлет. На самом деле при возникновении ошибки контейнер предоставляет некую полезную информацию, которую сервлет может использовать для генерирования ответа. Эта информация доступна в виде одного из атрибутов запроса; чтобы ее извлечь, вам понадобится метод `getAttribute()`:

```
Object requestAttribute = servletRequest.getAttribute("name");
```

Атрибуты во многом похожи на параметры сервлета, с той лишь разницей, что с их помощью можно передавать любые объекты. Мы уже сталкивались с ними, когда рассматривали контекст `ServletContext` в разделе «Интерфейс ServletContext» в этой главе. В этом случае речь идет об атрибутах запроса. Когда сервлет (или JSP-сценарий, или фильтр) вызывается для обработки ошибки, в его запросе находится следующий набор строковых значений:

```
javax.servlet.error.servlet_name
javax.servlet.error.request_uri
javax.servlet.error.message
```

В зависимости от того, каким образом объявлена ошибка (в виде кода или исключения), в запросе может быть доступен один из двух дополнительных атрибутов:

```
// целочисленный код или объект Exception
javax.servlet.error.status_code
javax.servlet.error.exception
```

В атрибуте `status_code` хранится код ошибки типа `Integer`. Атрибут `exception` содержит экземпляр исключения, который представляет ошибку. Индексные страницы для URL-адресов, которые ведут к директориям, можно назначать похожим образом. Обычно, когда пользователь пытается открыть директорию, веб-сервер ищет в ней документ, который должен выводиться по умолчанию. Самым распространенным примером такого документа является файл *index.html*. Вы можете объявить свой собственный упорядоченный список, в котором будет выполняться поиск нужной страницы. Для этого в файле *web.xml* необходимо указать тег `<welcome-file-list>`:

```
<welcome-file-list>
   <welcome-file>index.html</welcome-file>
   <welcome-file>index.htm</welcome-file>
</welcome-file-list>
```

Благодаря тегу `<welcome-file-list>` сервер понимает, что при запросе к директории нужно искать по очереди файлы под названием *index.html* и *index.htm*. Если ни один из них не будет найден, сервер сам должен решить, какую страницу нужно вывести. Обычно в этом случае выводится содержимое директории или сообщение об ошибке.

Безопасность и аутентификация

Одной из самых сильных сторон технологии Servlet API в контексте развертывания веб-приложений является возможность использования декларативного стиля для определения ограничений, связанных с безопасностью. Это означает, что прямо в файле *web.xml* можно определить, какие участки вашей программы (URL-адреса документов, директорий, сервлетов и т. д.) защищены паролем, какие пользователи имеют к ним доступ, а также класс протокола безопасности, необходимый для установления связи. Для реализации этих основных ограничений вовсе не обязательно добавлять какой-либо код в свои сервлеты.

В файле *web.xml* можно использовать два типа элементов, которые отвечают за безопасность и аутентификацию.

Теги `<security-constraint>` предоставляют механизм авторизации, основанный на ролях пользователей, и безопасный протокол передачи данных, если таковой потребуется. В теге `<login-config>` описывается тип аутентификации, применяемый в веб-приложении.

Защита ресурсов с помощью ролей безопасности

Давайте рассмотрим простой пример. В следующем фрагменте файла *web.xml* определяется участок под названием «Секретные документы» с шаблоном URL-адреса `/secret/*`; доступ к нему будут иметь только пользователи с ролью «secretagent». В нем также определен простейший механизм аутентификации, `BASIC`, с помощью которого в браузере выводится простое диалоговое окно для ввода имени пользователя и пароля:

```
<web-app>
...
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Секретные документы</web-resource-name>
         <url-pattern>/secret/*</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>secretagent</role-name>
      </auth-constraint>
   </security-constraint>
   <login-config>
      <auth-method>BASIC</auth-method>
   </login-config>
```

Каждый блок `<security-constraint>` содержит один раздел `<web-resource-collection>`, в котором указывается именованный список URL-шаблонов для определенных участков веб-приложения, и тег `<auth-constraint>` с пользовательскими ролями, имеющими доступ к этим участкам.

То же самое можно сделать при помощи аннотации `ServletSecurity` с элементом `HttpConstraint` внутри:

```
@ServletSecurity(
   @HttpConstraint(rolesAllowed = "secretagent")
)
public class SecureHelloClient extends HttpServlet
{ ...
```

Вы можете добавить эту аннотацию в свой тестовый сервлет или скопировать пример XML-кода в свой файл *web.xml* из архива *learningjava.war*. Но чтобы это сработало, вам придется сделать еще кое-что, а именно: создать в серверной среде своего приложения роль «secretagent» и самого пользователя с этой ролью.

Доступ к защищенным участкам выдается не отдельным пользователям, а ролям. Роль — это, в сущности, абстрактная группа пользователей. Каждый пользователь может иметь несколько ролей.

Вся информация (имена, пароли и т. д.) обрабатывается за пределами сервлета, в среде сервера приложений (который, в свою очередь, может быть интегрирован с операционной системой). Обычно серверы приложений имеют собственные инструменты для создания пользователей и назначения им ролей (как по отдельности, так и группами).

При попытке получить доступ к защищенной части приложения контейнер проверит, имеет ли пользователь соответствующую роль. Сервер Tomcat позволяет легко добавлять тестовых пользователей и назначать им роли — для этого достаточно отредактировать файл *conf/tomcat-users.xml*. Например, чтобы создать пользователя с именем «bond» и ролью «secretagent», нужно добавить следующую запись:

```
<user username="bond" password="007" roles="secretagent"/>
```

То, как управлять пользователями и ролями безопасности в других серверах, должно быть описано в соответствующей документации.

Защищенная передача данных

Прежде чем идти дальше, давайте рассмотрим еще один аспекте механизма безопасности — защищенность передачи данных. В конце каждого блока `<security-constraint>` должен содержаться тег `<user-data-constraint>`, который устанавливает один из трех уровней безопасности при обмене информацией с защищенным участком приложения. Например:

```
<security-constraint>
...
   <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

Помимо `CONFIDENTIAL` доступны значения `NONE` и `INTEGRAL`. Уровень `NONE` не выдвигает никаких особенных требований к передаче данных; того же эффекта можно добиться и без тега `<user-data-constraint>`. Это означает, что информация будет передаваться по сети в обычном текстовом виде. Уровень безопасности `INTEGRAL` требует, чтобы используемый протокол гарантировал неизменность данных при передаче. Это подразумевает использование цифровых подписей или других методик проверки принимаемых данных; при этом канал связи может оставаться открытым, без шифрования. Наконец, уровень `CONFIDENTIAL` требует все то же самое, что `INTEGRAL`, но с обязательным шифрованием данных. На практике единственным широко распространенным и безопасным средством обмена информацией в веб-браузерах является технология SSL. Обычно она используется на всех уровнях, кроме `NONE`.

Мы можем обеспечить защищенную передачу данных и в самом сервлете, используя аннотацию `ServletSecurity` с элементом `HttpMethodConstraint` внутри:

```
@ServletSecurity(
   httpMethodConstraints = @HttpMethodConstraint( value="GET",
      transportGuarantee = ServletSecurity.TransportGuarantee.
CONFIDENTIAL)
)
public class SecureHelloClient extends HttpServlet { ... }
@ServletSecurity(
   value = @HttpConstraint(rolesAllowed = "secretagent"),
   httpMethodConstraints = @HttpMethodConstraint( value="GET",
      transportGuarantee = ServletSecurity.TransportGuarantee.
CONFIDENTIAL)
)
public class SecureHelloClient extends HttpServlet { ... }
```

С помощью атрибута `httpMethodConstraints` и аннотации `HttpMethodConstraint` мы разрешаем доступ к сервлету только посредством запросов типа GET, отправленных по протоколу HTTP, и только с уров-

нем безопасности CONFIDENTIAL. В этом примере также демонстрируется использование уровней безопасности в сочетании с аннотацией rolesAllowed.

Аутентификация пользователей

В этом разделе вы научитесь создавать собственную веб-форму для аутентификации пользователей. Сначала мы рассмотрим декларативный подход на примере файла *web.xml*, а затем перейдем к более гибкому варианту, который предоставляет технология Servlet 3.0.

Раздел <login-conf> полностью определяет то, каким образом пользователь идентифицирует себя при входе в защищенный участок приложения. Тег <auth-method> позволяет выбрать один из четырех методов аутентификации: BASIC, DIGEST, FORM и CLIENT-CERT. В нашем примере использовался метод BASIC, при котором веб-браузер выводит стандартное диалоговое окно для ввода логина и пароля. При этом реквизиты пользователя передаются по сети в виде обычного текста (если только в теге <transport-guarantee> не было отдельно активировано шифрование потока данных с помощью протокола SSL). Метод DIGEST похож на BASIC, но используется не так часто; вся его защищенность ограничивается скрытием текста пароля. Еще один аналог метода BASIC, FORM, позволяет использовать для ввода логина и пароля собственную HTML-форму вместо стандартного диалога. Страница формы может быть как статической, так и сгенерированной с помощью сервлета. Но опять же, если отдельно не включить поддержку SSL, данные будут передаваться в текстовом виде. На этом фоне особый интерес представляет метод CLIENT-CERT, который требует, чтобы пользователь идентифицировался посредством клиентского сертификата открытого ключа. Это подразумевает использование протокола SSL (или подобного), который обеспечивает безопасный обмен данными и взаимную аутентификацию на основе цифровых сертификатов. Принцип установки клиентского сертификата зависит от конкретного веб-браузера.

Метод FORM является самым полезным, так как он позволяет изменять внешний вид страницы входа в систему (при этом мы рекомендуем использовать протокол SSL для защиты потока данных). Мы также можем указать документ, который будет выведен в результате неудачной аутентификации. Вот пример блока <login-config> с использованием метода FORM:

```
<login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/login_error.html</form-error-page>
   </form-login-config>
</login-config>
```

HTML-форма, размещенная на странице входа в систему, должна содержать поля для ввода логина и пароля, названные специальным образом. Вот простой пример файла *login.html*:

```
<html>
<head><title>Вход в систему</title></head>
<body>
   <form method="POST" action="j_security_check">
      Логин: <input type="text" name="j_username"><br>
      Пароль: <input type="password" name="j_password"><br>
      <input type="submit" value="Отправить">
   </form>
</body>
</html>
```

Поля для ввода логина и пароля называются, соответственно, `j_username` и `j_password`, а в качестве URL-адреса в атрибуте `action` указано значение `j_security_check`. К странице, предназначенной для вывода ошибок, никаких особых требований не выдвигается, но обычно на ней находится ссылка «попробовать еще раз», которая позволяет вернуться к форме для входа в систему.

В версии Servlet API 3.0 интерфейс `HttpServletRequest` содержит специальные методы для прохождения и отмены аутентификации. Однако в этом случае пользователь остается в системе только до тех пор, пока не закончится его сессия. Таким образом, чтобы отменить аутентификацию пользователя, достаточно вызвать из его сессии метод `invalidate()`:

```
request.logout(); request.getSession().invalidate();
```

С помощью технологии Servlet API 3.0 можно самостоятельно управлять процессом входа в систему, используя метод `login()` из объекта `ServletRequest`. Для этого достаточно подготовить сервлет, который будет принимать логин и пароль (в защищенном режиме), вызывая в конце метод `login()`. Это позволит вам полностью контролировать

условия прохождения аутентификации. И, конечно же, вы можете вывести пользователя из системы с помощью метода `logout()`.

```
@ServletSecurity(
   httpMethodConstraints = @HttpMethodConstraint( value="POST",
      transportGuarantee = ServletSecurity.TransportGuarantee.
CONFIDENTIAL)
)
@WebServlet( urlPatterns={"/mylogin"} )
public class MyLogin extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse
response)
      throws ServletException, IOException
   {
      String user = request.getParameter("user");
      String password = request.getParameter("pass");
      request.login( user, password );
      // Возвращаем ответ или перенаправляем на другую страницу...
   }
```

Авторизация в императивном стиле

Нужно сказать, что помимо декларативного механизма безопасности, реализованного в виде файла *web.xml*, существует и императивный (или процедурный) стиль, благодаря которому сервлеты могут самостоятельно использовать все данные об аутентификации, доступные через контейнер. Мы остановимся на некоторых основных его моментах, не углубляясь в подробности.

Имя аутентифицированного пользователя можно получить с помощью метода `getRemoteUser()` из объекта `HttpServletRequest`, а метод `getAuthType()` служит для определения типа аутентификации. Для работы с ролями безопасности предусмотрен метод `getAuthType()` (но чтобы его использовать, нужно выполнить дополнительное связывание в файле *web.xml*, которое позволит сервлету ссылаться на роли безопасности с помощью специальных псевдонимов).

Более продвинутые приложения могут извлекать из запроса объект `java.security.Principal`, используя метод `getUserPrincipal()`. Если данные передаются по безопасному протоколу, такому как SSL, метод

`isSecure()` возвращает `true`, а подробную информацию о том, каким образом пользователь был аутентифицирован (вид шифрования, длину ключа, цепочку сертификатов), можно получить с помощью атрибутов запроса. Стоит отметить, что с точки зрения контейнера сервлета понятие «нахождения в системе» определяется значением, которое возвращает метод `getUserPrincipal()` (для аутентифицированных пользователей оно не должно равняться `null`).

Фильтры сервлетов

Технология Java Servlet API предоставляет универсальный интерфейс `Filter`, который позволяет обрабатывать запросы и ответы сервлета с помощью модульных компонентов («фильтров»), объединенных в нечто вроде конвейера. Применяя сразу несколько фильтров, вы формируете последовательную цепочку, каждый элемент которой, прежде чем передавать запрос дальше, может его прочитать или изменить. Перед окончательной отправкой на клиентскую сторону через аналогичную цепочку может быть пропущен и ответ сервлета. Фильтры можно применять к любым запросам, поступающим в веб-приложение, а не только к тем, которые обрабатываются сервлетами; статические данные тоже можно фильтровать. Разработчик сам решает, нужно ли применять фильтры к индексным страницам, к сообщениям об ошибках, а также к страницам, которые передаются или подключаются в результате взаимодействия нескольких сервлетов.

Фильтры можно объявлять и привязывать к сервлетам в файле *web.xml* или посредством аннотаций. Привязка осуществляется двумя способами: либо с помощью URL-шаблонов (таких, которые используются в сервлетах), либо путем задания имени сервлета в конфигурационном файле. В обработке запроса участвуют все фильтры, которые соответствуют определенному пути, но в остальном здесь действуют те же правила сопоставления URL-адресов, что и в случае с сервлетами.

Фильтры вызываются в том порядке, в котором они указаны в файле *web.xml*; при этом теги `<url-pattern>` имеют приоритет над элементами `<servlet-name>`. В сервлетах все происходит наоборот — первыми обрабатываются точные совпадения. Объединение фильтров в цепочки происходит следующим образом: сначала вызываются те из них, которые имеют подходящие URL-шаблоны, затем в ход идут фильтры, связанные с именем текущего сервлета (в обоих случаях порядок их вызова

определяется файлом *web.xml*). Совпадение с URL-адресом имеет более высокий приоритет, нежели привязка к сервлету, поэтому в данном случае первым в очереди на обработку поступившего запроса может стоять фильтр с шаблоном /*.

Объявлять и привязывать фильтры к сервлетам можно также с помощью аннотации `WebFilter`, однако такой способ не позволяет определять порядок их вызова. Вы можете комбинировать декларативный и императивный стили, описывая в файле *web.xml* исключительно привязки (конфигурация фильтров будет рассмотрена позже в этой главе).

Интерфейс `javax.servlet.Filter`, который реализуют фильтры, очень простой и во многом перекликается с технологией Servlet API. Он содержит три метода: `init()`, `doFilter()` и `destroy()`. Метод `doFilter()` — это то место, где выполняется основная работа. Ему передаются все объекты `ServletRequest` и `ServletRespons`, которые поступают в сервлет. Прежде чем отправлять эти объекты следующему фильтру и, в конце, самому сервлету (или пользователю), мы можем их проанализировать и внести свои изменения — или даже подставить вместо них собственные экземпляры. В качестве третьего параметра для метода `doFilter()` выступает объект `FilterChain`, который служит ссылкой на оставшуюся часть цепочки. С его помощью вы можете вызвать следующий элемент конвейера. Пример фильтра представлен в следующем разделе.

Простой фильтр

Для начала давайте сделаем кое-что простое, но в то же время практичное: создадим фильтр, который ограничивает количество одновременных подключений по связанным с ним URL-адресам. Мы просто будем подсчитывать число активных соединений, проходящих через фильтр, и отвергать новые запросы при достижении заданного лимита:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;
public class ConLimitFilter implements Filter
{
   int limit;
   volatile int count;
   public void init( FilterConfig filterConfig )
```

```
        throws ServletException
    {
        String s = filterConfig.getInitParameter("limit");
        if ( s == null )
            throw new ServletException("Отсутствует начальный параметр:
"+limit);
        limit = Integer.parseInt( s );
    }
    public void doFilter (
        ServletRequest req, ServletResponse res, FilterChain chain )
            throws IOException, ServletException
    {
        if ( count > limit ) {
            HttpServletResponse httpRes = (HttpServletResponse)res;
            httpRes.sendError( httpRes.SC_SERVICE_UNAVAILABLE, "Занято.");
        } else {
            ++count;
            chain.doFilter( req, res );
            --count;
        }
    }
    public void destroy() { }
}
```

Класс `ConLimitFilter` реализует интерфейс `Filter` вместе с тремя его методами для управления жизненным циклом: `init()`, `doFilter()` и `destroy()`. В методе `init()` мы используем объект `FilterConfig`, чтобы найти начальный параметр с именем «`limit`» и превратить его в целое число. Вы можете установить это значение в одном из разделов файла *web.xml*, где объявляются наши фильтры (можно также воспользоваться аннотациями, как мы делали раньше). Метод `doFilter()` реализует всю нашу бизнес-логику. Получив из входящего запроса объекты `ServletRequest` и `ServletResponse`, он проверяет счетчик и либо передает их дальше по цепочке следующему фильтру, либо отвергает, генерируя собственный ответ. Во втором случае используется стандартное для протокола HTTP сообщение «504 Service Unavailable».

Вызывая метод `doFilter()` из объекта `FilterChain`, мы передаем управление следующему фильтру в цепочке (или сервлету, если фильтров больше не осталось). Если вызов нужно отклонить, мы генерируем наш собственный ответ с помощью объекта `ServletResponse` и позволя-

ем методу `doFilter()` завершить свою работу. На этом цепочка обработки обрывается, хотя любой фильтр, вызванный ранее, может вмешаться в этот процесс, пока запрос возвращается на клиентскую сторону.

Обратите внимание: наш фильтр `ConLimitFilter` увеличивает и уменьшает счетчик, соответственно, до и после вызова метода `doFilter()`. Прежде чем переходить к следующему фильтру (или сервлету), у нас есть возможность обработать запрос. После вызова `doFilter()` цепочка, ведущая к сервлету, заканчивается, и запрос возвращается клиенту. На этом этапе мы получаем возможность выполнить завершающую обработку нашего ответа.

Напоследок нужно сказать, что в качестве запроса и ответа метод `doFilter()` принимает не объекты `HttpServletRequest` и `HttpServletResponse`, которые мы тут упоминали, а их более обобщенные версии, `ServletRequest` и `ServletResponse`. При реализации фильтра необходимо определить, имеют ли эти объекты какое-то отношение к протоколу HTTP, и можно ли их привести к более конкретному типу (как раз это мы и сделали в нашем примере, чтобы иметь возможность отправлять сообщения об ошибках в рамках стандарта HTTP, используя метод `sendError()`).

Тестовый сервлет

Прежде чем идти дальше, давайте рассмотрим простой тестовый сервлет. На его примере вы сможете опробовать этот и другие фильтры, которые мы напишем в данном разделе. Как понятно из имени нашего класса, `WaitServlet`, его главная задача — просто ждать (wait). Время ожидания (в секундах) можно указать с помощью параметра `time` (это упрощенная версия класса `BackgroundWaitServlet`, который мы создали ранее в этой главе, когда обсуждали асинхронные сервлеты).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class WaitServlet extends HttpServlet
{
   public void doGet( HttpServletRequest request,
      HttpServletResponse response )
      throws ServletException, IOException
   {
```

```
String waitStr = request.getParameter("time");
if ( waitStr == null )
   throw new ServletException("Отсутствует параметр time");
int wait = Integer.parseInt(waitStr);

try {
Thread.sleep( wait * 1000 );
} catch( InterruptedException e ) {
throw new ServletException(e);
}
response.setContentType("text/html");
PrintWriter out = response.getWriter();
      out.println(
         "<html><body><h1>Ответ сервлета WaitServlet</h1></body></html>");
      out.close();
   }
}
```

Вы можете опробовать фильтр `ConLimitFilter` в работе, выполнив множество одновременных запросов к сервлету `WaitServlet`. Стоит заметить, что некоторые веб-браузеры не позволяют отправлять параллельные запросы по одному и тому же URL-адресу; кроме того, при открытии нескольких вкладок может возникнуть задержка. Чтобы обмануть веб-браузер, можно добавлять к запросу лишние параметры. Можно также воспользоваться утилитой командной строки *curl*, если она у вас установлена.

Объявление и привязка фильтров

Объявление и привязка фильтров в файле *web.xml* выполняется уже знакомым нам способом. Как и в случае с сервлетами, каждый экземпляр фильтра имеет отдельную запись, которая выглядит следующим образом:

```
<filter>
   <filter-name>defaultsfilter1</filter-name>
   <filter-class>RequestDefaultsFilter</filter-class>
</filter>
```

Здесь указан псевдоним фильтра, с помощью которого на него будут ссылаться в рамках файла *web.xml*, а также имя его класса. По анало-

гии с сервлетом, объявление фильтра может содержать элементы <init-param> с различными параметрами.

Чтобы привязать фильтр к ресурсу, используется тег <filter-mapping>, в котором вместе с псевдонимом фильтра указывается либо URL-шаблон (как мы уже видели раньше), либо псевдоним сервлета:

```
<filter-mapping>
   <filter-name>conlimitfilter1</filter-name>
   <servlet-name>waitservlet1</servlet-name>
</filter-mapping>
<filter-mapping>
   <filter-name>conlimitfilter1</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Аналогичным образом объявлять и привязывать фильтры позволяет аннотация WebFilter. Для выполнения привязки принимается один из двух атрибутов: urlPatterns или servletNames. Кроме того, эта аннотация поддерживает начальные параметры.

Фильтрация запроса к сервлету

Наш первый фильтр получился довольно скучным, так как он не вносил никаких изменений в информацию, которую принимал или отправлял наш сервлет. Давайте же займемся непосредственно «фильтрацией»: попробуем модифицировать входящий запрос до того, как он будет принят сервлетом. В этом примере мы создадим фильтр, который будет автоматически добавлять стандартные значения для указанных параметров сервлета, если они не содержатся во входящем запросе. Вот кода класса RequestDefaultsFilter:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestDefaultsFilter implements Filter
{
   FilterConfig filterConfig;
   public void init( FilterConfig filterConfig ) throws ServletException
   {
      this.filterConfig = filterConfig;
```

```
    }
    public void doFilter (
        ServletRequest req, ServletResponse res, FilterChain chain )
            throws IOException, ServletException
    {
        WrappedRequest wrappedRequest =
            new WrappedRequest( (HttpServletRequest)req );
        chain.doFilter( wrappedRequest, res );
    }
    public void destroy() { }
    class WrappedRequest extends HttpServletRequestWrapper
    {
        WrappedRequest( HttpServletRequest req ) {
            super( req );
        }
        public String getParameter( String name ) {
            String value = super.getParameter( name );
            if ( value == null )
                value = filterConfig.getInitParameter( name );
            return value;
        }
    }
}
```

Чтобы вклиниться в процесс передачи данных, нам пришлось сделать нечто виртуозное. Мы «похитили» входящий запрос `HttpServletRequest`, поставив на его место свой «лжеобъект», который подчиняется нашим приказам. Чтобы модифицировать запрос (а позже и ответ), мы помещаем его внутрь специального адаптера, что позволяет нам переопределить некоторые его методы. В этом примере мы перехватываем управление методом `getParameter()`, чтобы он не просто возвращал `null`, а пытался найти значения по умолчанию.

Мы опять реализовали три метода для управления жизненным циклом, но на этот раз, прежде чем делать вызов `doFilter()`, чтобы перейти к следующему фильтру в цепочке, мы помещаем входящий запрос `HttpServletRequest` внутрь нашего собственного объекта `WrappedRequest`. Класс `WrappedRequest` наследует специальный адаптер под названием `HttpServletRequestWrapper`, который является производной класса `HttpServletRequest` и играет роль удобной обертки. Он принимает ссылку на объект `HttpServletRequest` и по умолчанию делегирует ему все по-

ступающие вызовы. Это очень удобно, потому как мы можем переопределять только те методы, которые нас интересуют. В классе `WrappedRequest` нам достаточно переопределить метод `getParameter()`, добавив в него нужную функциональность. В этом примере мы просто вызываем родительскую версию `getParameter()`; если она возвращает значение `null`, мы пытаемся найти в фильтре начальный параметр с тем же именем.

Запустите этот пример вместе с сервлетом `WaitServlet`. Ниже приведен фрагмент из файла *web.xml*, предназначенный для объявления и привязки нашего фильтра, а также аннотация, которая делает то же самое:

```
<filter>
   <filter-name>defaultsfilter1</filter-name>
   <filter-class>RequestDefaultsFilter</filter-class>
   <init-param>
      <param-name>time</param-name>
      <param-value>3</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>defaultsfilter1</filter-name>
   <servlet-name>waitservlet1</servlet-name>
</filter-mapping>
@WebFilter(
   servletNames = "waitservlet1",
   initParams = {
      @WebInitParam(name="time", value="3")
   }
)
```

Теперь сервлет `WaitServlet` будет получать стандартное значение временного интервала (3 секунды), даже если вы сами забудете его указать.

Фильтрация ответа, который возвращается сервлетом

Процесс фильтрации запросов оказался довольно простым. Используя тот же подход, мы можем отфильтровать и ответ. Для этого результат работы сервлета перед отправкой на клиентскую сторону нужно поместить внутрь объекта `HttpServletResponseWrapper`. Это позволит нам перехватывать вызовы методов, которые используются сервлетом для записи ответа (то же самое мы делали с методом `getParameter()`, пред-

назначенным для чтения входящих данных). Например, мы можем переопределить метод `sendError()` из объекта `HttpServletResponse`, сделав так, чтобы он перенаправлял пользователя на заданную страницу. Так мы могли бы эмулировать механизм управления выводом ошибок, который предлагает файл *web.xml*. Но у нас есть возможность сделать нечто куда более интересное, и это мы продемонстрируем в данном разделе. Речь идет об изменении данных, записанных сервлетом, до того, как они дойдут до клиента. Для этого нам придется пойти на еще более сложные ухищрения. Мы поместим ответ сервлета в обертку, чтобы переопределить метод `getWriter()`, после чего создадим еще одну обертку для клиентского объекта `PrintWriter`, который возвращается этим методом; с ее помощью мы будем буферизировать и изменять записываемые данные. Несмотря на свою действенность и практичность, этот прием может оказаться довольно запутанным.

В следующем примере под названием `LinkResponseFilter` показан фильтр для автоматической генерации гиперссылок. Он считывает ответы, отправляемые в формате HTML, и ищет в них определенные строки с помощью регулярных выражений. Найденные значения превращаются в гиперссылки. Регулярные выражения и ссылки нужно указывать в виде начальных параметров фильтра. Вы можете сделать из этого примера удобную систему для управления веб-сайтом, добавив дополнительные условия и поддержку баз данных или XML-файлов. Вот код фильтра:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class LinkResponseFilter implements Filter
{
   FilterConfig filterConfig;
   public void init( FilterConfig filterConfig )
      throws ServletException
   {
      this.filterConfig = filterConfig;
   }
   public void doFilter (
      ServletRequest req, ServletResponse res, FilterChain chain )
         throws IOException, ServletException
   {
      WrappedResponse wrappedResponse =
```

```
            new WrappedResponse( (HttpServletResponse)res );
        chain.doFilter( req, wrappedResponse );
        wrappedResponse.close();
    }
    public void destroy() { }
    class WrappedResponse extends HttpServletResponseWrapper
    {
        boolean linkText;
        PrintWriter client;
        WrappedResponse( HttpServletResponse res ) {
            super( res );
        }
        public void setContentType( String mime ) {
            super.setContentType( mime );
            if ( mime.startsWith("text/html") )
                linkText = true;
        }
        public PrintWriter getWriter() throws IOException {
            if ( client == null )
                if ( linkText )
                    client = new LinkWriter(
                        super.getWriter(), new ByteArrayOutputStream() );
                else
                    client = super.getWriter();
            return client;
        }
        void close() {
            if ( client != null )
                client.close();
        }
    }
    class LinkWriter extends PrintWriter
    {
        ByteArrayOutputStream buffer;
        Writer client;
        LinkWriter( Writer client, ByteArrayOutputStream buffer ) {
            super( buffer );
            this.buffer = buffer;
            this.client = client;
        }
        public void close() {
```

```
        try {
            flush();
            client.write( linkText( buffer.toString() ) );
            client.close();
        } catch ( IOException e ) {
            setError();
        }
    }
    String linkText( String text ) {
        Enumeration en = filterConfig.getInitParameterNames();
        while ( en.hasMoreElements() ) {
            String pattern = (String)en.nextElement();
            String value = filterConfig.getInitParameter( pattern );
            text = text.replaceAll(
                pattern, "<a href="+value+">$0</a>" );
        }
        return text;
    }
}
```

Этот пример получился немного длиннее предыдущего, но суть осталась прежней. Мы поместили наш запрос `HttpServletResponse` внутрь объекта `WrappedResponse`, используя вспомогательный класс `HttpServletResponseWrapper`. Класс `WrappedResponse` переопределяет два метода: `getWriter()` и `setContentType()`. Наша версия метода `setContentType()` устанавливает флаг, сигнализирующий о том, что исходящие данные имеют тип «`text/html`» (то есть, это HTML-документ). Через наш фильтр могут проходить двоичные файлы (например, изображения), и было бы неразумно с нашей стороны выполнять для них замену с помощью регулярных выражений. Мы также переопределили метод `getWriter()`, чтобы предоставить наш собственный исходящий поток, `LinkWriter`, который является производной класса `PrintStream`. Он принимает клиентские объекты `PrintWriter` и `ByteArrayOutputStream`, играющие роль буфера, в который данные попадают непосредственно перед записью. Здесь мы действуем достаточно осторожно, подменяя поток только в том случае, если флаг `linkText` из метода `setContent()` равен `true`. Если подмена все же произошла, поток начинает кешироваться, чтобы каждый последующий вызов метода `getWriter()` возвращал один и тот же экземпляр `LinkWriter`. В конце мы добавили метод `close()`, которого нельзя найти в обычном запросе типа `HttpServletResponse`. Мы используем его

во время возвращения данных клиенту, сигнализируя о том, что объект `LinkWriter` должен прекратить работу и отправить окончательный ответ. Это делается на тот случай, если при завершении обработки сервлет так и не дождался закрытия исходящего потока со стороны клиента.

Итак, мы рассмотрели все важные аспекты нашего примера с записывающим фильтром. Теперь давайте обратим внимание на объект `LinkWriter`, в котором и происходит вся магия. Это производная класса `PrintStream`, которая хранит ссылки на два других потока: клиентский `PrintWriter` и промежуточный `ByteArrayOutputStream` . Объект `LinkWriter` вызывает конструктор родительского класса и передает ему экземпляр `ByteArrayOutputStream`, чтобы всего стандартные методы `print()` выполняли запись в байтовый массив. Нам остается только перехватывать метод `close()` из объекта `PrintStream` и добавлять ссылки в текст перед отправкой данных. Когда поток `LinkWriter` закрывается, он передает все данные, буферизированные в его родительском классе, в объект `ByteArrayOutputStream`. Затем он извлекает их оттуда (с помощью метода `toString()`), вызывает свой метод `linkText()`, чтобы добавить в них гиперссылки, и возвращает клиенту. Метод `linkText()` просто перебирает все начальные параметры фильтра, воспринимая их как шаблоны для регулярных выражений, и превращает их в гиперссылки, используя метод `String.replaceAll()` (больше о методе `replaceAll()` можно узнать в главе 1).

Несмотря на свою работоспособность, этот пример имеет определенные ограничения. Во-первых, мы не можем буферизировать произвольный объем данных. Более продуманная реализация могла бы сама решать, когда нужно начинать передачу информации клиенту; для этого можно было бы позволить клиенту определять размер буфера в объекте `HttpServletResponse`. Во-вторых, наша версия метода `linkText()` могла бы быть более быстрой, если бы мы сформировали одно общее регулярное выражение с использованием чередования. Несомненно, вы и сами сможете найти массу способов улучшить наш пример.

Создание WAR-файлов с помощью утилиты Ant

До этого момента мы не сильно озадачивались подбором специального инструментария для создания своих приложений. С одной стороны, эта книга имеет немного другую тематику, с другой — ее автор не очень любит привязывать себя к конкретным средам разработки. Тем не менее есть один универсальный инструмент, который должен входить

в арсенал любого программиста на платформе Java: утилита Ant, созданная в рамках проекта Jakarta. Этот сборщик проектов, полностью написанный на языке Java, играет ту же роль, что команда `make` в мире языка C. В отличие от `make`, утилита Ant специально заточена для сборки Java-кода — не в последнюю очередь благодаря богатому набору «целей» (декларативных команд) для выполнения различных операций, связанных с платформой Java (таких как создание WAR-файлов). Это быстрый, переносимый и простой в использовании инструмент, который, к тому же, легко устанавливается. Вам определенно стоит с ним подружиться.

Мы не станем рассматривать его во всех подробностях. На домашней странице утилиты Ant* можно загрузить установочный пакет и получить всю необходимую информацию. Чтобы вас заинтересовать, мы покажем вам сборочный файл, который вы можете найти среди других примеров для этой главы. Он позволит вам скомпилировать собственный исходный код и собрать из него полноценный WAR-файл.

Иерархия директорий, ориентированная на разработку

В начале этой главы мы уже описывали структуру WAR-архива, включая стандартные файлы и директории, которые он должен содержать. И хотя такая организация необходима для развертывания приложения в архиве, она не очень подходит для ведения проекта во время разработки. Хранение файла *web.xml* и библиотек внутри директории *WEB-INF*, находящейся на уровень выше остальной части приложения, может быть удобным при использовании команды *jar*, но с точки зрения программиста такое расположение не очень хорошо подходит для создания и поддержки всех этих компонентов. К счастью, с помощью утилиты Ant и простого сборочного файла мы можем создать WAR-архив из проекта с любой структурой.

Давайте выберем иерархию директорий, которая будет более удобной на этапе разработки. Например:

```
myapplication
|
|-- src
|-- lib
|-- docs
|-- web.xml
```

* **jakarta.apache.org/ant/**

Директории *src*, *lib* и *doc* отводятся для хранения, соответственно, исходного кода, библиотек и ресурсов. Файл *web.xml* мы оставили на верхнем уровне, чтобы было легче изменять параметры и т. д.

Ниже представлен простой пример файла *build.xml*, предназначенного для формирования WAR-архива из нашей новой иерархии директорий:

```
<project name="myapplication" default="compile" basedir=".">
   <property name="war-file" value="${ant.project.name}.war"/>
   <property name="src-dir" value="src" />
   <property name="build-dir" value="classes" />
   <property name="docs-dir" value="docs" />
<property name="webxml-file" value="web.xml" />
<property name="lib-dir" value="lib" />
<target name="compile" depends="">
   <mkdir dir="${build-dir}"/>
   <javac srcdir="${src-dir}" destdir="${build-dir}"/>
</target>
<target name="war" depends="compile">
   <war warfile="${war-file}" webxml="${webxml-file}">
      <classes dir="${build-dir}"/>
      <fileset dir="${docs-dir}"/>
      <lib dir="${lib-dir}"/>
   </war>
</target>
<target name="clean">
   <delete dir="${build-dir}"/>
   <delete file="${war-file}"/>
   </target>
</project>
```

Точно такой же файл вы можете найти в примерах исходного кода для этой главы. Чтобы скомпилировать свой проект, запустите команду *ant* без параметров (цель по умолчанию). Если вы хотите создать WAR-архив, укажите в качестве цели `war`:

```
% ant war
```

Файл **build.***xml* содержит инструкции, согласно которым утилита Ant должна найти все исходные java-файлы в директории *src*, скомпилировать их и поместить результат в каталог *classes*. Команда *ant war* создает архив *myapplication.war* с правильно структурированными ресурсами

и файлом *web.xml*. Утилита *antclean* позволяет удалить все сгенерированные элементы, включая директорию *classes* и WAR-архив.

Этот сборочный файл совершенно не связан с каким-либо конкретным проектом, если не считать атрибута `name` в первой строчке, в котором вы можете указать название своего приложения (хотя оно будет использоваться исключительно в имени генерируемого архива). Изменяя элементы `<property>`, вы можете скорректировать имена файлов и директорий, чтобы они соответствовали структуре вашего проекта. Данная версия сборочного файла *build.xml* содержится в архиве *learningjava.war*, который вы можете найти среди прочих примеров для этой главы.

Развертывание WAR-архивов с помощью утилиты Ant

Вместе с сервером Tomcat можно загрузить клиентский пакет под названием «deployer», который предоставляет различные цели для утилиты Ant, включая развертывание (начальное и повторное), запуск, остановку и удаление веб-приложения. Этот пакет использует возможности встроенной в Tomcat системы управления контейнерами. Похожую интеграцию с утилитой Ant предоставляют и другие серверы, такие как WebLogic. Используя эти инструменты в своих сборочных скриптах, вы сможете сэкономить множество времени и усилий. Пакет deployer можно загрузить на веб-сайте SourceForge*.

Реализация веб-сервисов

Раз уж мы разобрались с сервлетами и веб-приложениями, можно вернуться к теме веб-сервисов. В предыдущей главе мы говорили о веб-сервисах, как дополнении к основным HTTP-транзакциям, в котором документы формата XML не потребляются веб-браузером на клиентской стороне, а используются для взаимодействия между приложениями. Тогда мы показывали, насколько просто выглядит вызов функций из веб-сервиса, написанного в стиле технологии RPC, если при этом использовать клиентские классы, сгенерированные на основе WSDL-документа. В этом разделе мы сосредоточимся на серверной стороне и продемонстрируем процесс реализации и развертывания веб-сервиса.

---

* **sourceforge.net/projects/tomcatdeployer/files/TomcatDeployer/1.0/**

Мир веб-сервисов быстро развивается; появляются новые API-интерфейсы и модные термины, не отстает и маркетинг. Перспективы этого подхода к межпрограммному взаимодействию, основанного на простых веб-протоколах, немного омрачает излишняя бюрократизация процесса разработки новых стандартов, а также попытки конкурирующих компаний добавить в концепцию веб-сервиса различные возможности и целые уровни. Но правда заключается в том, что раньше простота и изящество веб-сервисов (особенно в сравнении с более замысловатыми протоколами) достигались, в основном, за счет их скудной функциональности; они не поддерживали управление состоянием, функции обратного вызова, транзакции и аутентификацию, не обеспечивали безопасность. Вместе с добавлением всех этих возможностей концепция усложнялась. Мы не станем подробно рассматривать все аспекты этой обширной темы, а сосредоточим свое внимание на веб-сервисах, основанных на технологии RPC, которые подходят для разработки широко спектра простых приложений.

В главе 14 мы подробно прошлись по процессу генерирования и запуска клиентской стороны веб-сервиса (метеослужбы). В этот раз мы создадим и развернем наш собственный веб-сервис, EchoService, который просто будет возвращать клиенту любые входящие параметры. Для запуска этого примера мы воспользуемся контейнером и API-интерфейсами, которые предоставляет технология JAX-WS, хотя, если упаковать его в WAR-архив и немного сконфигурировать, он может быть развернут и на сервере Tomcat.

Объявление сервиса

В главе 14 мы начали создание нашего клиентского API-интерфейса с загрузки WSDL-файла, содержащего описание веб-сервиса (тогда это была метеослужба). Напомним, что формат WSDL основан на XML и предназначен для описания функций сервиса и типов данных, которые он принимает и возвращает. Из этого описания команда *wsimport* смогла сгенерировать клиентские классы, которые были необходимы для взаимодействия с удаленным сервисом на платформе Java.

При создании собственного веб-сервиса у нас есть как минимум два варианта. Мы можем воспользоваться аналогичным подходом, написав новый WSDL-документ. На его основе утилита *wsimport* сгенерирует все необходимые серверные API-интерфейсы, помеченные аннотациями, которые мы потом сможем самостоятельно реализовать. Но есть куда более простой путь: начать с написания кода.

У утилиты *wsimport* есть антипод под названием *wsgen*. Эта команда может считывать помеченные аннотациями классы, генерируя на их основе WSDL-документ и прочие компоненты сервиса. Она сама позаботится о создании всех этих файлов, если мы развернем наш класс при помощи инструментов для конечной публикации, которые предоставляет технология JAX-WS. Это означает, что для проверки работоспособности простого веб-сервиса нам достаточно написать класс, методы которого будут помечены подходящими аннотациями, и затем опубликовать этот код. Проще не бывает.

Наш веб-сервис Echo

Мы напишем простой сервис, который будет принимать и возвращать обратно несколько разных видов значений: `int`, `String` и наш собственный объектный тип (предназначенный для хранения данных), `MyObject`. В следующем разделе мы более подробно остановимся на типах данных и процессе их обработки. А пока что займемся кодом:

```
package learningjava.service;
import javax.jws.*;
import javax.xml.ws.Endpoint;
@WebService
public class Echo
{
   @WebMethod
   public int echoInt( int value ) { return value; }
   @WebMethod
   public String echoString( String value ) { return value; }
   @WebMethod
   public MyObject echoMyObject( MyObject value ) { return value; }
   public static void main( String[] args )
   {
      Endpoint endpoint = Endpoint.publish( "http://localhost:8080/echo",
         new Echo() );
   }
}
public class MyObject
{
   int intValue;
   String stringValue;
   public MyObject() { }
```

```
    public MyObject( int i, String s ) {
        this.intValue = i;
        this.stringValue = s;
    }
    public int getIntValue() { return intValue; }
    public void setIntValue( int intValue ) { this.intValue = intValue; }
    public String getStringValue() {
        return stringValue;
    }
    public void setStringValue( String stringValue ) {
        this.stringValue = stringValue;
    }
}
```

Для каждого типа была предусмотрена отдельная операция с префиксом «echo», так как стандарт WSDL не поддерживает перегрузку методов (В случае конфликта имен API-интерфейс JAX-WS выводит предупреждение во время выполнения программы и сам выбирает подходящий метод.) Мы поместили наш код в пакет `learningjava.service`, потому что это упростит работу со сторонними инструментами. Выбранное нами название будет использоваться в стандартном *пространстве имен*, а также в качестве имени пакета для сгенерированного клиентского кода. Чтобы его изменить, можно воспользоваться атрибутом `targetNamespace` для аннотации `WebService` (это действительно хорошее решение, которое позволит повысить стабильность нашего интерфейса).

Для развертывания нашего сервиса мы используем метод `publish()` из класса `Endpoint`, предоставляемого технологией JAX-WS. Вместе с идентификатором URI, который состоит из имени сервера, порта и пути к сервису, этот метод принимает экземпляр нашего класса. Очевидно, что единственным сервером, который нам подходит, является наш локальный компьютер; обычно доступ к нему можно получить по имени «localhost». Наш сервис работает на порту 8080 и имеет путь */echo*.

Использование сервиса

После запуска веб-сервиса откройте его URL-адрес в своем веб-браузере, чтобы получить тестовую страницу. Если запуск происходит на том же компьютере, адрес будет совпадать с идентификатором URI, который вы передали в метод `publish()`. Хотя в некоторых случаях «localhost» нужно заменить на «127.0.0.1».

```
http://localhost:8080/echo
http://127.0.0.1:8080/echo
```

Перед вами должно появиться описание сервиса, как показано на рис. 15.1. Это будет свидетельствовать о том, что сервис активен и готов предоставить доступ к своей конфигурационной информации. Вы можете щелкнуть мышью по ссылке в разделе «WSDL», чтобы просмотреть WSDL-файл, который был сгенерирован для нашего сервиса. Адрес этой ссылки должен быть таким же, как у тестовой страницы, только с «**?wsdl**» на конце.

С помощью WSDL-файла можно сгенерировать клиентское приложение и протестировать наш сервис, как мы это делали в главе 14. Ниже представлена команда, благодаря которой сгенерированные классы помещаются в отдельный пакет с именем `learningjava.client.impl`, чтобы не спутать их с нашим собственным кодом. Мы также использовали ключ `-keep`, который позволяет сохранять не только скомпилированные class-файлы, но и исходный код (возможно, вам захочется на него взглянуть). В качестве последнего аргумента в этой команде выступает URL-адрес сгенерированного нами WSDL-файла, который вы можете найти на тестовой странице (см. выше).

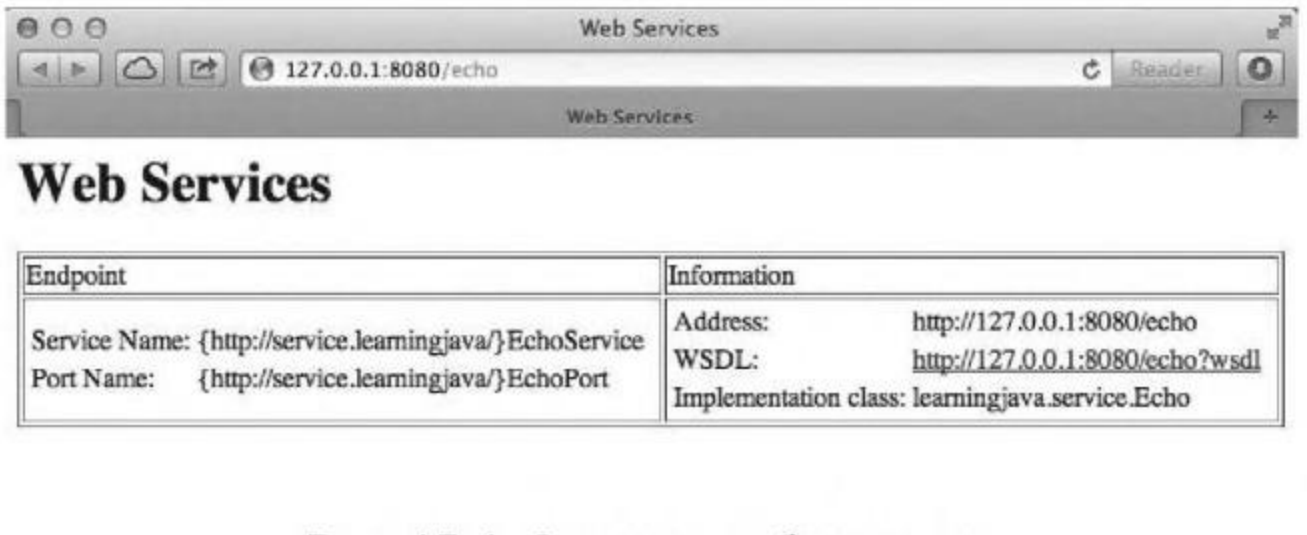

Рис. 15.1. Описание веб-сервисов

```
% wsimport -p learningjava.client.impl -keep http://localhost:8080/echo?wsdl
```

Теперь, чтобы протестировать наш сервис, создадим небольшой клиент на основе сгенерированных классов:

```
package learningjava.client;
import learningjava.client.impl.*;
public class EchoClient
{
   public static void main( String [] args ) throws java.rmi.RemoteException
```

```
{
    Echo service = new EchoService().getEchoPort();
    int i = service.echoInt( 42 );
    System.out.println( i );
    String s = service.echoString( "Привет!" );
    System.out.println( s );
    MyObject myObject = new MyObject();
    myObject.setIntValue( 42 );
    myObject.setStringValue( "Строка" );
    MyObject myObj = service.echoMyObject( myObject );
    System.out.println( myObj.getStringValue() );
}
```

Как видно из этого кода, утилита *wsimport* сгенерировала класс `EchoService`, который представляет наш сервис. Такие классы могут быть разделены на несколько групп, поэтому, чтобы получить интерфейс `Echo`, мы запрашиваем его «порт» с помощью метода `getEchoPort()` (в контексте технологии WSDL термин *порт* обозначает интерфейс сервиса).

После запуска клиента значения отправляются на сервер, откуда они должны вернуться обратно; результат будет выведен на экран. Вот и все! Как было сказано в самом начале этой главы, для реализации и запуска сервиса требуется минимальный объем кода. Платформа Java идеально подходит для работы с веб-сервисами, так как в ее стандартной поставке уже содержится подходящий контейнер.

Типы данных

Как вы уже могли догадаться, существуют некоторые ограничения относительного того, какими типами объектов можно обмениваться с веб-сервисом. Дело в том, что для описания данных используется стандартный формат XML. Технологии JAX-WS и WSDL имеют встроенную поддержку большинства простых типов и классов, которые часто применяются на платформе Java. Если точнее, эта поддержка реализована на уровне API-интерфейсов библиотеки JAXB (архитектура привязки данных к XML-документам), которые используются внутри технологии JAX-WS. Подробней об этой библиотеке и о языке XML Schema мы поговорим в главе 24.

Интерфейсы JAX-WS и JAXB позволяют разбирать классы, которые предназначены для передачи данных и совместимы с технологией

JavaBeans (при условии, что они состоят из поддерживаемых типов). Это позволяет передавать собственные объекты, что и было продемонстрировано в сервисе `Echo` на примере аргумента `MyObject`.

Стандартные типы

В табл. 15.1 собраны все типы, которые поддерживаются в интерфейсах JAX-WS напрямую (те, что упоминаются непосредственно в спецификации языка XML Schema; больше об описании типов данных в XML-документах можно прочитать в главе 24).

Табл. 15.1. Стандартные типы

Категория	Типы
Простые типы и их обертки	`boolean`, `Boolean`, `byte`, `Byte`, `short`, `Short`, `float`, `Float`, `int`, `Integer`, `long`, `Long`, `double`, `Double`
Классы	`java.lang.String`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Calendar`, `java.util.Date`, `java.util.UUID`, `java.net.URI`, `java.awt.Image` (в виде `byte []`)
Коллекции	Массивы (`Array`), списки (`List`) и наборы (`Set`)

На сегодняшний день ассоциативные массивы и другие сложные типы коллекций не поддерживаются. Чтобы обеспечить более широкую совместимость при разработке кроссплатформенных веб-сервисов, стоит формировать свои объекты из простых типов данных, а также из массивов и списков этих типов.

Объекты-значения

Как уже было сказано, технология JAX-WS может работать с объектами нестандартных типов, хотя при этом существует ряд требований и оговорок. Во-первых, чтобы объект можно было передавать, его поля должны иметь поддерживаемые типы (или их сочетания). Во-вторых, объект обязан соблюдать два шаблона проектирования из спецификации JavaBeans: у него должен быть публичный конструктор без аргументов, и любые его частные поля должны иметь геттеры и сеттеры. Более подробно эти вопросы освещены в главе 22.

Напоследок стоит упомянуть, что в отличие от технологии Java RMI веб-сервисы не поддерживают передачу «поведения» и не обеспечивают идентичность передаваемых объектов. Классы, которые генерируются на основе WSDL-документа, содержат всего лишь копию тех данных,

которые хранятся в указанных нами компонентах. Такие «объекты-значения» будут передаваться вместе со всем содержимым наших настоящих объектов, не имея при этом никакого отношения к оригиналу. Тем не менее данные, которые получит наша серверная реализация, будут иметь вид «настоящих» объектов, ничем не отличающихся от тех, что остались на клиенте. Вот для чего нам нужен конструктор без аргументов: чтобы серверное приложение само смогло создать и инициализировать объект, с которым оно потом будет работать.

Итог

Данная глава посвящена веб-приложениям и веб-сервисам — одной из самых обширных и динамичных сфер программирования, которые рассматривались в этой книге. Было бы невозможно охватить эту тему полностью, поэтому мы обсуждали ее в контексте поддержки со стороны платформы Java. Для расширения кругозора мы рекомендуем вам воспользоваться дополнительной литературой; особенно это касается разработки приложений на основе технологий HTML5 и JavaScript, которые способны взаимодействовать с сервлетами и веб-сервисами, написанными на языке Java.

Глава 16
БИБЛИОТЕКА SWING

Swing — это набор инструментов для построения графического пользовательского интерфейса на языке Java. Пакет `javax.swing` (и бесчисленное множество его дочерних пакетов) содержит классы, представляющие такие элементы интерфейса как окна, кнопки, раскрывающиеся и древовидные списки, таблицы и меню — все, что нужно для создания современных, функционально насыщенных клиентских приложений.

Классы Swing входят в состав библиотеки под названием Java Foundation Classes (JFC), которая содержит следующие API-интерфейсы:

- Abstract Window Toolkit (AWT) — исходная библиотека для разработки пользовательского интерфейса, предоставляющая основные графические классы.
- Swing — библиотека для создания пользовательского интерфейса, полностью написанная на языке Java.
- Инструменты, предназначенные для интеграции в пользовательский интерфейс нестандартных средств ввода/вывода.
- 2D API — богатый набор классов для создания высококачественной двумерной графики.
- API-интерфейсы для поддержки «Drag and Drop» — управление графическими элементами пользовательского интерфейса путем их перетаскивания.

Библиотека JFC является одним из крупнейших и наиболее сложных компонентов платформы Java, поэтому нет ничего удивительного в том, что на ее рассмотрение уйдет сразу несколько глав. На самом деле мы затронем только самые важные ее части — Swing и 2D API. План дальнейших действий таков:

- В этой главе мы покроем основные концепции построения пользовательского интерфейса с помощью библиотеки Swing.
- Глава 17 посвящена базовым элементам интерфейса: кнопкам, спискам, полям ввода, флажкам и т. д.

- В главе 18 мы продолжим изучать инструменты, которые предлагает библиотека Swing, подробно останавливаясь на классах для работы с текстом и древовидными структурами, таблицах и других продвинутых элементах интерфейса.
- В главе 19 рассматриваются диспетчеры компоновки, которые отвечают за размещение элементов интерфейса внутри окна.
- Глава 20 охватывает основы создания двумерной графики, в том числе простой вывод изображений.
- В главе 21 вы научитесь генерировать и обрабатывать изображения с помощью инструментов, входящих в состав пакета `java.awt.image`, а также познакомитесь с аудио- и видеоформатами.

Мы не в состоянии охватить все возможности библиотеки Swing в рамках этой книги; если вы хотите изучить эту технологию от начала и до конца, вам стоит ознакомиться с книгой «Java Swing» авторства Марка Лойя, Роберта Экштейна, Дэйва Вуда, Брайана Коула и Джеймса Эллиотта (издательство O'Reilly). Мы же сосредоточимся на основных инструментах, которые с высокой долей вероятности пригодятся вам в будущем. Здесь также будет продемонстрировано применение некоторых наиболее продвинутых возможностей этой библиотеки. На рис. 16.1 показаны компоненты пользовательского интерфейса, которые входят в состав пакета `javax.swing`.

Чтобы лучше понять концепцию технологии Swing, стоит обратиться к ее предшественнице — библиотеке AWT (Abstract Window Toolkit). Уже только по названию понятно, что это некая абстракция. Как и вся платформа Java, она проектировалась с прицелом на кроссплатформенность, чтобы предоставлять одни и те же возможности вне зависимости от реализации виртуальной машины. Обычно пользователи и разработчики хотят, чтобы внешний вид и поведение приложения вписывались в систему, на которой оно выполняется, хотя у всех систем свои требования. Библиотека AWT предоставляет одинаковую функциональность на всех платформах, но при этом обеспечивает сходство со стандартными программами в каждом конкретном случае. Идея в том, что приложение, которое разрабатывается для операционной системы Windows, должно своим внешним видом и поведением вписываться (более или менее) в такие среды как X Window System или Macintosh. Привязка к конкретной платформе достигается за счет взаимозаменяемых «слоев», которые для вывода компонентов пользовательского интерфейса взаимодействуют с текущей оконной системой. Это позволяет Java-приложениям абстрагироваться от нюансов среды,

в которой они выполняются, не засоряя свои API-интерфейсы лишними сущностями. Допустим, вы попросили библиотеку AWT создать для вас кнопку. При запуске приложения или апплета один из слоев этой библиотеки рисует кнопку, которая будет выглядеть так, как любая другая кнопка на текущей платформе, будь то Windows, Macintosh или что-то другое.

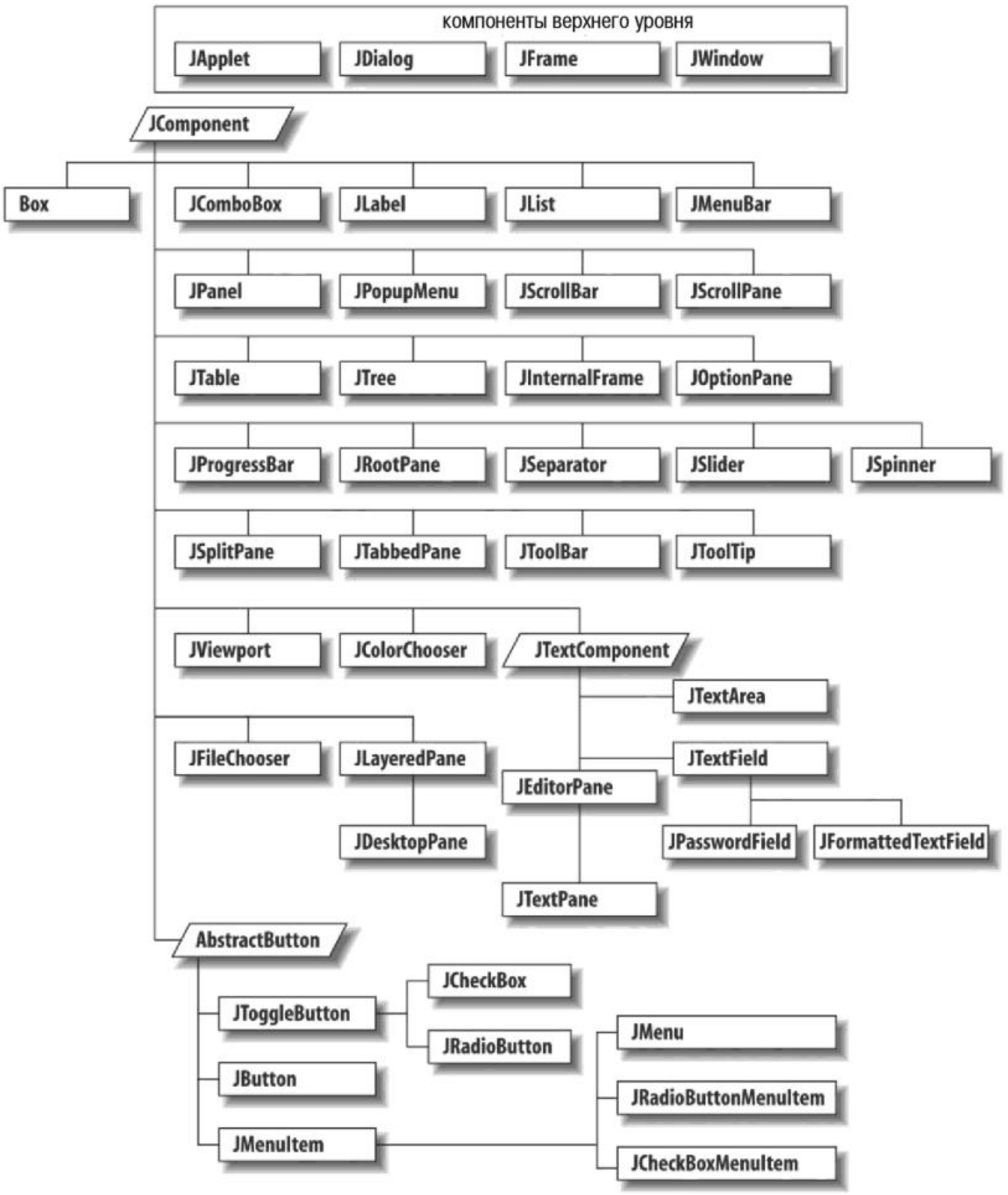

Рис. 16.1. Элементы пользовательского интерфейса в пакете `javax.swing`

Однако библиотека AWT всегда обладала некоторыми серьезными недостатками. Хуже всего было то, что ввиду использования системно-

го инструментария одно и то же приложение могло вести себя немного по-разному на разных платформах. Более того, если какой-то сложный элемент пользовательского интерфейса (например, древовидный список или таблица) не имел аналогов в некоторых системах, его реализация не попадала в состав AWT. Эта библиотека обеспечивала подходящие внешний вид и поведение графических приложений, написанных на языке Java, но их функциональность и переносимость были ограничены.

В библиотеке Swing используется принципиально иной подход. Такие элементы интерфейса, как кнопки и раскрывающиеся списки, она реализует самостоятельно, а не заимствует из системы. Это означает, что, к примеру, кнопка будет выглядеть одинаково в любой среде. При этом библиотека Swing предоставляет мощный, расширяемый API-интерфейс для изменения внешнего вида, который позволяет эмулировать визуальный стиль операционной системы на уровне платформы Java. Реализация всех элементов на одном языке позволила значительно снизить зависимость от ошибок, связанных с конкретной средой, что являлось большой проблемой для библиотеки AWT. Это также обеспечило высокий уровень гибкости и расширяемости пользовательского интерфейса, недоступный для системных компонентов.

Библиотека Swing проектировалась таким образом, чтобы упростить работу с графическими элементами. При создании пользовательского интерфейса в вашем распоряжении будет широкий выбор готовых компонентов (кнопок, полей ввода), которые можно легко скомпоновать внутри контейнеров. При необходимости вы сможете создавать на их основе совершенно новые переносимые элементы, пригодные к многократному использованию.

В библиотеке Swing за размещение *компонентов* внутри *контейнеров*, а также за изменение их размеров местоположения отвечают *диспетчеры компоновки*. Расстановка компонентов определяется не точными значениями, а некоей универсальной стратегией. Например, вы можете описать пользовательский интерфейс с набором кнопок и полей ввода, который всегда будет выглядеть корректно даже при изменении размеров окна. И неважно, на какой платформе работает приложение, и какой визуальный стиль вы используете — за правильное положение компонентов друг относительно друга отвечает диспетчер компоновки.

В следующих двух главах приводятся примеры использования большинства компонентов из пакета `javax.swing`. Но прежде чем приступить

к их обсуждению, нам нужно познакомиться с принципом создания и управления пользовательским интерфейсом, который исповедуется в библиотеке Swing. Этот материал поможет вам понять концепции, лежащие в основе графических интерфейсов, и то, как они используются на платформе Java.

Компоненты

Фундаментальным объектом пользовательского интерфейса на платформе является *компонент*. Из компонентов состоит любое графическое Java-приложение. Речь идет о таких вещах, как окна, панели, кнопки, флажки, полосы прокрутки, списки, меню и поля ввода. Обычно перед использованием компонент следует поместить в *контейнер*. Контейнер группирует компоненты у себя внутри, размещает их с помощью диспетчера компоновки и связывает их с конкретным устройством вывода. Как видно на рис. 16.1, все компоненты в библиотеке Swing имеют общего прародителя — абстрактный класс `javax.swing.JComponent`. Например, кнопка `JButton` наследует класс `AbstractButton`, который, в свою очередь, является производной класса `JComponent`.

`JComponent` является корневым элементом в иерархии классов библиотеки Swing, но при этом он сам унаследован от класса `Container`, который входит в состав AWT. Вот так выясняется, что на самом низком уровне библиотека Swing берет свое начало в пакете AWT. Класс `Container` происходит от базового компонента библиотеки AWT, `Component`, а тот является производной класса `Object`. В связи с этим объекты типа `JComponent` обладают возможностями как компонента, так и контейнера.

Тем не менее библиотеки AWT и Swing имеют параллельные иерархии. В первом случае корневым элементом является класс `Component` , а во втором — `JComponent`. Оба пакета содержат похожие классы (например, `Button` и `JButton` , `List` и `JList`), но в отличие от AWT библиотека Swing поддерживает более развитые элементы управления и может предложить реализацию парадигмы MVC (Model-View-Controller — модель-представление-контроллер), о чем которой мы поговорим позже.

Чтобы вам было легче понять назначение класса `JComponent`, разделим его функции на две категории: внешний вид и поведение. Он содержит методы и переменные для управления отображением объекта. Речь идет о таких базовых атрибутах как видимость, текущие размер и положение, а также о графических установках — шрифте и фоновом цвете,

которые в разных дочерних классах используются по-разному. В классе `JComponent` присутствуют методы для работы с графикой и событиями; их переопределяют во время наследования, чтобы получить всевозможные разновидности виджетов (вскоре вы сами в этом убедитесь).

При отображении компонент привязывается к определенному устройству вывода. Класс `JComponent` инкапсулирует доступ к отведенной ему области на экране. Он включает в себя инструменты для отрисовки графики, работы с невидимыми ресурсами и получения пользовательского ввода. Внутри него активно используются интерфейсы 2D API для выполнения таких операций, как сглаживание шрифтов, оптимизация графического вывода, отображение подсказок. В последних версиях платформы Java (6 и выше) скорость и качество отрисовки во многих случаях не уступают аналогичным показателям в приложениях, которые скомпилированы под конкретную операционную систему.

Говоря о поведении графических компонентов, мы имеем в виду то, как они реагируют на события, вызванные пользователем. Когда пользователь выполняет некое действие внутри области вывода компонента (например, щелкает мышью по кнопке), поток выполнения библиотеки Swing генерирует объект, который описывает произошедшее событие. Этот объект передается компонентам, подписавшимся на отслеживание событий этого типа. К примеру, когда пользователь щелкает по кнопке, генерируется экземпляр `ActionEvent`. Чтобы получать экземпляры этого класса, объект должен зарегистрироваться в соответствующем компоненте (кнопке) и реализовать интерфейс `ActionListener`.

События доставляются путем вызова соответствующих методов-обработчиков внутри принимающих объектов (их еще называют «слушателями»). С помощью методов интерфейса `ActionListener` (например, `actionPerformed()`) эти объекты принимают события тех типов, которые их интересуют. Каждый тип отвечает за определенный вид взаимодействия с компонентом. Например, `MouseEvent` описывает действия, связанные с указателем мыши в рамках области вывода, `KeyEvent` сигнализирует о нажатиях клавиш на клавиатуре, а высокоуровневые события (такие как `ActionEvent`) сообщают о том, что компонент пользовательского интерфейса выполнил свою работу.

На событиях основана вся работа пользовательского интерфейса на платформе Java, поэтому мы постараемся изучить их во всех подробностях уже в этой главе. Хотя применение событий не ограничивается сугубо графическими задачами; это важный механизм взаимодействия

между объектами, который можно использовать в совершенно разных компонентах приложения. Особую роль они играют в контексте технологии JavaBeans, где с их помощью доставляются уведомления о произошедших изменениях.

Событийная модель библиотеки Swing является очень гибкой. Она не требует, чтобы каждый компонент отслеживал и обрабатывал события на своем участке пользовательского интерфейса. Программист может зарегистрировать произвольное количество «обработчиков», которые будут получать уведомления от одного или нескольких компонентов, интегрируя их в бизнес-логику приложения. Контейнер, к примеру, обрабатывает некоторые события, связанные с его дочерними элементами.

В мире графических приложений главная обязанность контейнера заключается в размещении содержащихся в нем компонентов так, чтобы они визуально не выходили за его границы. Если компонент делает нечто такое, что может затронуть его «соседей» (например, изменяет свой размер или видимость), он сообщает об этом контейнеру. Контейнер же связывается с диспетчером компоновки и просит его перегруппировать дочерние компоненты.

Как мы уже упоминали, все компоненты библиотеки Swing являются производными класса `Container`. Это не означает, что все они могут играть роль контейнера для любых других элементов интерфейса. Отношения между этими сущностями определяются на более низком уровне. Чтобы хранить и управлять объектами `JComponent`, контейнеру не нужно знать, что они собой представляют и чем они занимаются. Компоненты можно легко переставлять и заменять новыми версиями, формировать из них новые составные объекты пользовательского интерфейса, которые сами потом будут восприниматься как отдельные компоненты. Это может пригодиться при разработке более сложных элементов управления, которые подходят для многократного использования.

Системные компоненты и внешний вид

Компоненты библиотеки Swing часто называют *самодостаточными* или *легковесными*. Речь идет о той связи с системными инструментами для отображения элементов интерфейса, которая есть у технологии AWT (и которой нет у Swing). Чтобы вывести на экран стандартные компоненты, библиотека AWT использует набор объектов, которые связывают чистый код на языке Java с текущей операционной системой.

Конечно, любые компоненты должны вызывать системные методы, чтобы как-то взаимодействовать с внешней средой; вопрос только в том, на каком уровне это происходит. Библиотека AWT использует связующие интерфейсы из пакета `java.awt.peer`, с помощью которых графические объекты, полностью написанные на языке Java, могут использовать соответствующие системные компоненты. Но все эти интерфейсы и объекты инкапсулированы внутри класса `Component`, так что вам не нужно взаимодействовать с ними напрямую.

В библиотеке AWT такое связывание используется очень активно. Например, если вы создадите окно и разместите на нем восемь кнопок, для вас автоматически будет создано девять связующих объектов (восемь для кнопок и один для окна). Как разработчику приложений вам не нужно беспокоиться об этих объектах — они находятся где-то глубоко внутри, выполняя свою работу по взаимодействию с оконными средствами операционной системы.

В отличие от AWT, в библиотеке Swing большинство компонентов являются *самодостаточными* или *легковесными*. Это означает, что они не имеют никакой прямой связи с исходной оконной системой. Они отображаются в своем родительском контейнере и реагируют на действия пользователя при помощи чистого Java-кода, не полагаясь на системные средства. В библиотеке Swing только окна самого верхнего уровня (которые имеют наиболее низкоуровневый API-интерфейс) взаимодействуют с внешней средой. Эти окна происходят от аналогичных компонентов из состава AWT и, следовательно, используют связующие интерфейсы. Иными словами, если создать окно и разместить на нем восемь кнопок, библиотека Swing сгенерирует только один связующий объект — для окна. Благодаря менее тесному взаимодействию с оконной системой (в сравнении с AWT), технология Swing не так сильно зависит от специфики той или иной платформы.

Легковесные компоненты позволяют легко изменять внешний вид пользовательского интерфейса. Ввиду своей автономности они сами могут решать, каким образом их нужно выводить на экран. В связи с этим библиотека Swing поддерживает разные визуальные стили, которые можно менять «на лету» (речь идет о едином внешнем виде компонентов приложения). Разработчику доступно несколько визуальных стилей. Некоторые из них являются оригинальными (Metal, Synth и Nimbus), а другие эмулируют внешний вид разных оконных окружений, среди которых Windows, Macintosh и Motif. Для кроссплатформенных приложений стандартным стилем является Metal. Он выглядит плоским, ми-

нималистическим и, в то же время, очень функциональным. Однако на фоне современных настольных систем он кажется устаревшим. Стиль Synth позволяет видоизменять оформление программ с помощью XML-файлов и графических ресурсов. В десятом обновлении для платформы Java SE 6 появился стиль Nimbus, который с эстетической точки зрения ничем не уступает современным операционным системам, таким как OS X и Windows. В нем используется векторная графика, что позволяет легко адаптировать компоненты интерфейса под ультрасовременные мониторы с высокой плотностью точек. Если вы хотите, чтобы ваши приложения выглядели одинаково на всех платформах, стиль Nimbus подойдет для этого как нельзя лучше.

Архитектура MVC в пользовательских интерфейсах

Прежде чем продолжать обсуждение различных концепций, связанных с пользовательским интерфейсом, мы бы хотели немного отвлечься и поговорить об архитектуре MVC. Как вы уже знаете, MVC — это модель разработки компонентов, пригодных для многократного использования, при которой структура, представление и поведение логически изолированы друг от друга. Этот принцип используется для решения множества разных задач проектирования, в том числе для построения элементов пользовательского интерфейса; это одна из основных концепций платформы Java.

Главная идея архитектуры MVC заключается в отделении модели данных элемента от способа его отображения. Например, в электронной таблице одну и ту же информацию можно выводить по-разному — в виде гистограмм, круговых диаграмм и т. д. Данные играют роль *модели*; способ их вывода можно считать *представлением*. У одной модели может быть множество разных представлений. За поведение компонента пользовательского интерфейса отвечает *контроллер*. Обычно он занимается внесением изменений в модель, что, в свою очередь, влияет на представление. Возьмем для примера элемент «флажок». В качестве его модели данных может выступать единственная переменная типа `Boolean`, которая определяет, установлен он или сброшен. Обработка щелчков мыши будет влиять на модель, а представление, прежде чем выводить компонент на экран, будет анализировать исходные данные.

То, как объекты библиотеки Swing взаимодействуют между собой, передавая события от источников к обработчикам, является частью концепции разделения, которая лежит в основе архитектуры MVC. Объек-

ты могут быть «наблюдателями» (контроллерами) и «наблюдаемыми» (моделями).* Первые занимаются обработкой событий, а вторые играют роль источников. Когда источник изменяется или выполняет какую-то операцию, он уведомляет об этом всех своих наблюдателей.

Библиотека Swing имеет встроенную поддержку MVC. Каждый ее компонент состоит из двух частей. Первая часть, которая называется делегатом пользовательского интерфейса, берет на себя роль «представления» и «контроллера». Она отвечает за отрисовку компонентов и реакцию на события, вызванные пользователем. Вторая часть является моделью данных. Благодаря такому разделению несколько компонентов могут иметь общую модель. Например, вы можете вывести один и тот же массив строк одновременно в текстовой метке и раскрывающемся списке.

Рисование

В таких библиотеках как Swing, где во главе угла стоит событийная модель, компоненты могут получить запрос на перерисовку в любой момент. Более императивные средства разработки обычно подразумевают, что компоненты перерисовываются либо при создании, либо при изменении внешнего вида. На платформе Java поведение компонентов тесно связано с состоянием графической среды. Например, если элемент управления перекрывается другим окном, а затем опять становится видимым, поток выполнения библиотеки Swing может попросить его выполнить перерисовку.

Чтобы инициировать перерисовку, из компонента вызывается метод `paint()`. Это можно сделать в любой момент, но чаще всего это происходит при первом появлении на экране, во время изменения внешнего вида или когда графическая система нечаянно «портит» область вывода компонента. Так как метод `paint()` обычно не имеет ни малейшего представления о причине своего вызова, он вынужден перерисовывать весь компонент целиком. Система может ограничить область рисования, если компонент требует только частичного обновления, но это уже не ваша забота. Библиотека Swing пытается действовать «умно», избегая перерисовки компонентов везде, где это возможно (в том числе за счет «промежуточного буфера»).

---

* В главе 11 мы познакомились с классом `Observer` и интерфейсом `Observable` из пакета `java.util`. Эти компоненты не задействованы напрямую в библиотеке Swing, но именно по их примеру организована вся работа источников и обработчиков событий.

Компонент никогда не перерисовывается самостоятельно. Вместо этого он вызывает метод `repaint()`, который, в свою очередь, просит у библиотеки Swing запланировать для него перерисовку. Через какое-то время метод `paint()` будет вызван извне. Библиотека Swing сама управляет этими вызовами, стараясь сделать их как можно более эффективными. Она понимает, что слишком большое количество запросов, особенно если сразу несколько из них адресовано одному и тому же компоненту, можно свести к единственному вызову метода `paint()`; вы можете быть уверены только в том, что при отправке запроса компонент будет перерисован как минимум один раз.

Обычно вызов метода `repaint()` подразумевает, что обновление должно произойти как можно скорее. Но вы также можете указать временной промежуток, в рамках которого вы бы хотели выполнить перерисовку; это позволяет системе более гибко планировать свои запросы. Попытка обновления произойдет до истечения указанного периода, но если в этот промежуток времени от вас поступило сразу несколько вызовов, система может объединить их в один. Приложения, которые выполняют простую анимацию, могут использовать данный подход для управления частотой смены кадров (где частота обратно пропорциональна заданному временному промежутку).

Как вы уже знаете, компоненты библиотеки Swing могут выступать в качестве контейнеров для других компонентов. И поскольку каждый элемент управления выводится самостоятельно, о необходимости выполнить перерисовку им обязан сообщать их контейнер. К счастью, все это делается автоматически внутри метода `paint()`. Но если вы собрались его переопределять, не забудьте делегировать вызов в родительский класс:

```
public void paint(Graphics g) {
   super.paint(g);
   ...
}
```

Есть и более простой путь решения этой проблемы. Любой компонент библиотеки Swing содержит метод `paintComponent()`, который, в отличие от метода `paint()`, отвечает только за собственную перерисовку. Вы можете переопределить только его, оставляя метод `paint()` нетронутым — это позволит вам не беспокоиться об обновлении любых вложенных компонентов.

Методы `paint()` и `paintComponent()` имеют один общий аргумент — объект `Graphics`. Он представляет графический контекст компонента

и привязан к участку экрана, на котором тот может быть нарисован. Кроме того, он содержит методы для рисования графических примитивов и обработки с изображений (мы еще вернемся к нему в главе 18).

Включение и отключение компонентов

Стандартные компоненты библиотеки Swing можно включать и отключать с помощью метода `setEnabled()`. Выключенный компонент становится «блеклым» и перестает реагировать на действия пользователя.

Давайте попробуем создать компонент, которым можно воспользоваться всего один раз. Для этого нам придется забежать немного вперед; позже мы рассмотрим все аспекты этого примера, которые сейчас вам могут показаться непонятными. Мы уже говорили, что при нажатии кнопка `JButton` генерирует событие `ActionEvent`, которое потом передается в метод `actionPerformed()`, принадлежащий слушателю. Следующий код отключает компонент, который сгенерировал это событие:

```
public boolean void actionPerformed(ActionEvent e ) {
   ((JComponent)e.getSource()).setEnabled(false);
}
```

Данный код вызывает метод `getSource()`, чтобы выяснить, какой именно компонент сгенерировал событие. Результат приводится к типу `JComponent`, так как нам неизвестен настоящий тип компонента, с которым мы имеем дело; событие типа `ActionEvent` могло быть сгенерировано не только кнопкой, но и любым другим элементом управления. В итоге компонент, ответственный за событие, отключается. Таким же образом вы можете отключить целый контейнер. Например, вместе с объектом `JPanel` отключаются все компоненты, которые он содержит.

Фокусируемся

Чтобы иметь возможность принимать события, связанные с клавиатурой, компонент должен *находиться в фокусе* — то есть, *быть выделенным* на экране (как логически, так и визуально). Такой компонент будет принимать всю информацию о событиях, отправленных с клавиатуры, до тех пор, пока фокус не сместится на другой объект. Обычно элементы управления выделяются при щелчке мышью или когда пользователь переходит к ним с помощью клавиатуры. Но они и сами могут запрашивать фокус, используя метод `requestFocus()` из класса `JComponent`. Разрешить или запретить выделение компонента можно с помощью метода

`setFocusable()`. Большинство компонентов, включая кнопки и раскрывающиеся списки, являются «фокусируемыми». Чтобы влиять на возможность установления фокуса для всего окна вместе со всем его содержимым, используйте метод `setFocusableWindowState()` из класса `Window`.

Обычно управление фокусом является для пользователя одним из важнейших аспектов взаимодействия с приложением. Особенно это касается полей ввода и форм — пользователи привыкли к удобной навигации при помощи клавиатурных клавиш (**Tab** и **Shift+Tab** для перехода к следующему или предыдущему полю). Управление фокусом может оказаться довольно сложной задачей, если речь идет о развесистом пользовательском интерфейсе с множеством компонентов. К счастью, с момента выхода платформы Java 1.4, в которой была полностью переделана подсистема фокусировки, библиотека Swing делает все это автоматически, поэтому программисту обычно не нужно самому заниматься логикой выделения элементов. Гибкий класс `KeyboardFocusManager` по умолчанию обеспечивает предсказуемую навигацию по содержимому окна, но вы можете влиять на нее с помощью объектов `FocusTraversalPolicy`. События, связанные с фокусировкой компонентов, мы обсудим позже в этой главе, а в главе 18 будет рассмотрена навигация с помощью клавиатуры.

Другие методы класса `JComponent`

Класс `JComponent` очень большой, так как он должен предоставлять базовый функционал для графических компонентов всех возможных разновидностей. Многие свои функции он унаследовал от родительских классов `Container` и `Component`. Вряд ли мы бы смогли найти место для всех его методов, но вам определенно стоит обратить свое внимание на некоторые наиболее важные из них:

```
Container getParent()
String getName()
void setName(String name)
```

Получают или назначают строковое имя для заданного компонента. Именование компонентов может пригодиться во время отладки. Назначенное имя можно получить с помощью метода `toString()`.

```
void setVisible(boolean visible)
```

Делает компонент видимым или невидимым в рамках его контейнера. При его использовании диспетчер компоновки автоматически расставляет видимые компоненты.

```
Color getForeground()
void setForeground(Color c)
void setBackground(Color c)
Color getBackground()
```

Получают и устанавливают цвета переднего и заднего плана для заданного компонента. Во всех элементах управления для рисования по умолчанию используется цвет переднего плана.

Например, с его помощью окрашивается текст в поле ввода и любые примитивы, нарисованные с помощью объекта `Graphics`, который передается в методы компонента, такие как `paint()` и `paintComponent()`. Фоновый цвет используется в стандартной реализации метода `update()` для заливки во время очистки компонента.

```
Dimension getSize()
void setSize(int width, int height)
```

Получают или устанавливают текущий размер компонента. Стоит заметить, что выбранный вами размер может быть откорректирован диспетчером компоновки. Чтобы изменить размер, которым компонент сам «хочет обладать», используйте метод `setPreferredSize()`. Класс `JComponent` также содержит методы для управления местоположением компонентов, но обычно этим занимается диспетчер компоновки.

```
Dimension getPreferredSize()
void setPreferredSize(Dimension preferredSize)
```

Используйте эти методы, чтобы узнать или установить компоненту предпочтительный размер. Это значение учитывается диспетчером компоновки при масштабировании элементов управления. После его изменения нужно вызвать метод `revalidate()`, чтобы откорректировать местоположение элементов.

```
Cursor getCursor()
void setCursor(Cursor cursor)
```

Получают или устанавливают внешний вид, который приобретает указатель мыши, находясь над областью компонента. Например:

```
JComponent myComponent = ...;
Cursor crossHairs =
   Cursor.getPredefinedCursor( Cursor.CROSSHAIR_CURSOR );
myComponent.setCursor( crossHairs );
```

Контейнеры

Контейнер — это такой компонент, который предназначен для хранения и управления другими компонентами. Существует три основных типа контейнеров: `JFrame`, `JPanel` и `JApplet`. Компонент `JFrame` представляет окно верхнего уровня и является производной класса `java.awt.Window`, который отличается от своего потомка только отсутствием рамки (в библиотеке Swing есть свой аналог класса `Window` — `JWindow`). `JPanel` — это универсальный контейнер, который группирует элементы управления внутри компонента `JFrame` и других экземпляров `JPanel`. Класс `JApplet` является фундаментом для апплетов, предназначенных для выполнения в рамках веб-браузера. Как и другие контейнеры, `JApplet` может хранить различные компоненты пользовательского интерфейса. Вы также можете использовать объект `JComponent` в качестве универсального контейнера, по примеру `JPanel`. Если не считать классы `JFrame`, `JWindow`, `JApplet` и `JDialog` (еще один вид оконного контейнера), берущие свое начало в пакете `java.awt`, все компоненты и контейнеры библиотеки Swing являются легковесными.

Контейнер ведет список «дочерних» компонентов, за которые он ответственен. Для управления ими у него предусмотрены специальные методы. Стоит сказать, что когда речь идет о родительских и дочерних компонентах, имеется в виду чисто визуальная связь, а не иерархия классов. Большинство элементов управления сами по себе являются довольно бесполезными; какой-то смысл они приобретают только после добавления в контейнер и отображения на экране. Чтобы добавить компонент в контейнер, нужно вызвать метод `add()` из класса `Container`. После этого его можно будет отобразить в рамках видимой области родительского объекта, управляя его местоположением с помощью диспетчера компоновки. Для удаления компонентов из контейнера служит метод `remove()`.

Диспетчеры компоновки

Диспетчер компоновки (или просто компоновщик) — это объект, который отвечает за размещение и масштабирование компонентов внутри области вывода контейнера. Определяя местоположение и размер компонентов, он чем-то похож на систему управления окнами. У любого контейнера есть стандартный компоновщик, но вы можете установить свой собственный диспетчер, воспользовавшись методом `setLayout()` из класса `Container`.

Библиотека Swing предоставляет несколько готовых версий диспетчера компоновки, которые реализуют основные схемы размещения ком-

понентов. В контейнере `JPanel` по умолчанию используется диспетчер `FlowLayout`, который пытается размещать элементы управления слева направо и сверху вниз, учитывая их предпочтительные размеры. Диспетчер `BorderLayout`, который применяется в классе `JFrame`, распределяет объекты по специальным областям окна, таким как `NORTH`, `SOUTH` и `CENTER`. Есть еще два компоновщика: `GridLayout` и `GridBagLayout`. Первый использует для размещения компонентов прямоугольную сетку, а второй, являясь наиболее универсальным, по своим возможностям напоминает HTML-таблицу. Каждый из этих диспетчеров будет подробно рассмотрен в главе 19.

Для добавления компонентов в контейнер со стандартным компоновщиком часто используется версия метода `add()`, которая принимает единственный аргумент типа `Component`. Но если вы используете такие диспетчеры компоновки как `BorderLayout` или `GridBagLayout`, вам необходимо указать дополнительную информацию о будущем местоположении своего компонента. Для этого в метод `add()` можно передать второй, уточняющий аргумент. Вот как можно разместить элемент управления в верхней части контейнера при условии, что в нем используется диспетчер `BorderLayout`:

```
myContainer.add(myComponent, BorderLayout.NORTH);
```

Здесь в качестве уточняющего объекта выступает статическое поле `NORTH`. Класс `GridBagLayout` использует куда более сложные параметры позиционирования.

Вставки

Вставка определяет толщину границ контейнера; это пространство не может быть использовано диспетчером компоновки. Вставка представлена объектом `Insets`, который содержит четыре публичных целочисленных поля: `top`, `bottom`, `left` и `right`. Обычно границы контейнера устанавливаются автоматически, и вам не нужно о них беспокоиться. При этом учитывается пространство вверху окна, которое может занимать строка меню. Чтобы получить размер вставки, вызовите из компонента метод `getInsets()`, который вернет объект `Insets`.

Порядок наложения компонентов

Стандартные компоновщики не позволяют компонентам перекрываться. Но это ограничение можно обойти, если использовать собственные диспетчеры или абсолютное позиционирование. В таком случае

начинает играть роль порядок, в котором компоненты добавляются в контейнер. При перекрытии они «накладываются» друг на друга: первый добавленный компонент находится вверху, а последний внизу. Для управления процессом наложения есть две специальные версии метода `add()`; они принимают дополнительный целочисленный аргумент, который позволяет задать точное положение элемента при наложении в рамках контейнера. И хотя в большинстве случаев вам вовсе не обязательно задумываться об этих деталях, сам факт наличия такой возможности является приятным.

Методы `revalidate()` и `doLayout()`

Диспетчер компоновки размещает компоненты в контейнере только в том случае, если его об этом попросить. Внешний вид скомпонованного контейнера можно «испортить» несколькими способами:

- Изменить его размер
- Изменить размер или переместить один из его дочерних компонентов
- Добавить, отобразить, удалить или спрятать дочерний компонент

В результате любого из этих действий контейнер будет помечен как *недействительный*. Другими словами, он должен заново расставить свои компоненты с помощью диспетчера компоновки. В большинстве случаев это делается автоматически на уровне библиотеки Swing. Любые элементы управления, не только контейнеры, хранят информацию о том, являются ли они действительными. При изменении размера, местоположения или внутренней компоновки объекта неизбежно вызывается метод `revalidate()`. Сначала недействительными становится сам компонент и все контейнеры, в которые он заключен; для этого используя метод `invalidate()`. Затем рекурсивно проверяется корректность расстановки всей иерархии компонентов, начиная с *корневого* контейнера. Чтобы проверить объект `Container`, из него вызывается метод `doLayout()`, который просит, чтобы диспетчер компоновки сделал свою работу; после этого контейнер снова становится действительным, так как он обновил свое содержимое. Корневым называют контейнер, который способен вместить в себя дочерний элемент сколь угодно большого размера (это может быть, например, объект `JScrollPane`). Такой контейнер может перенести любые возможные изменения в своей дочерней иерархии, не оказывая никакого влияния на собственные родительские компоненты.

Бывают ситуации, когда программисту приходится самому просить библиотеку Swing, чтобы та все исправила. К примеру, это может произойти при изменении предпочтительного размера компонента (в случае с реальным размером все происходит автоматически). Чтобы навести порядок в контейнере, вызовите из него метод `revalidate()`. Допустим, у вас есть некий объект `JPanel` (например, эмулятор клавиатуры с несколькими кнопками) и вы хотите изменить его предпочтительный размер с помощью метода `setPreferredSize()`. В таком случае вам нужно будет вызвать из него (или его непосредственного контейнера) метод `revalidate()`, чтобы диспетчер компоновки заново разместил кнопки на панели с учетом новых размеров.

Управление компонентами

Класс `Container` содержит еще несколько методов, о которых стоит упомянуть:

```
Component[] getComponents()
```

Возвращает компоненты контейнера в виде массива.

```
void list(PrintWriter out, int indent)
```

Генерирует список компонентов в заданном контейнере, записывая результат в поток `PrintWriter`.

```
Component getComponentAt(int x, int y)
```

Возвращает компонент, чье местоположение совпадает с заданными координатами в рамках системы координат контейнера.

Прослушивание компонентов

Интерфейс `ContainerListener` позволяет автоматизировать процесс отслеживания компонентов. Контейнер, который реализует этот интерфейс, сможет получать события о добавлении или удалении любого дочернего объекта.

JFrame и JWindow

Объекты `JWindow` и `JFrame` являются для компонентов контейнерами верхнего уровня. `JWindow` — это, в сущности, обычный графический прямоугольник, который отображается оконной системой. Он не имеет

обрамления, и поэтому лучше всего подходит для вывода всплывающих окон и элементов, которые выходят за рамки родительского контейнера (например, меню и раскрывающихся списков). Объект `JFrame`, являясь производной `JWindow`, имеет заголовок, кнопки управления окном (**Закрыть**, **Свернуть** и т. д.) и рамку. Используя стандартные средства оконной среды, вы можете перетаскивать этот компонент по экрану и изменять его размер. На рис. 16.2 слева показан объект `JFrame`, а справа объект `JWindow`.

Все остальные компоненты библиотеки Swing должны находиться внутри одного из этих объектов. Этому правилу обязаны следовать даже апплеты, которые сами являются разновидностью класса `Container`, хотя обычно они не имеют внешней рамки, так как выполняются внутри веб-браузера или программы *appletviewer*.



Рис. 16.2. `JFrame` и `JWindow`

Объекты `JFrame` и `JWindow` являются единственными компонентами, которым для отображения не нужен внешний контейнер. Чтобы вывести их на экран, достаточно вызвать метод `setVisible()`. Ниже представлено небольшое приложение, которое создает объекты `JFrame` и `JWindow`, и отображает их рядом друг с другом, в точности как на рис. 16.2.

```
//файл: TopLevelWindows.java
import javax.swing.*;
public class TopLevelWindows {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Объект JFrame");
      frame.setSize(300, 300);
      frame.setLocation(100, 100);
      JWindow window = new JWindow();
```

```
        window.setSize(300, 300);
        window.setLocation(430, 100);

        frame.setVisible(true);
        window.setVisible(true);
    }
}
```

Конструктор класса `JFrame` может принимать строковой аргумент, значение которого будет использоваться в качестве заголовка окна (вы также можете использовать конструктор без аргументов, а заголовок указать с помощью метода `setTitle()`). Размер и расположение окна на экране определяются методами `setSize()` и `setLocation()`. После `JFrame` мы точно таким же образом создали объект `JWindow`. Отличие лишь в том, что окно `JWindow` не имеет заголовка, поэтому его конструктор не принимает аргументов.

Подготовив объекты `JFrame` и `JWindow`, вы можете вывести их на экран, вызвав метод `setVisible(true)`. Этот метод отрабатывает мгновенно, не блокируя поток выполнения. Несмотря на то, что мы достигли последней инструкции в методе `main()`, наше приложение продолжает работать, оставаясь видимым. Чтобы закрыть окно `JFrame`, вы можете просто щелкнуть по кнопке **Закрыть** в его заголовке. При этом будет вызван метод `setVisible(false)`, а сам объект никуда не денется. Вы можете изменить такое поведение с помощью метода `setDefaultCloseOperation()` или добавив слушателя событий (но об этом мы поговорим позже). В этом примере мы не предусмотрели никаких средств для закрытия самого приложения, поэтому, чтобы остановить выполнение программы `TopLevelWindows`, нажмите **Ctrl+C** (или другую комбинацию клавиш, которая прерывает работу процесса на вашем компьютере).

В библиотеке AWT есть специальный класс под названием `Splash Screen`, который идеально подходит для вывода заставки во время запуска приложения. До выхода платформы Java 1.6 разработчикам приходилось использовать для этих целей объекты `Window` или `JWindow`, но это было компромиссным решением, так как данные компоненты отображаются только после запуска Java-машины и загрузки библиотек AWT и Swing. Новый вид заставки, `SplashScreen`, позволяет указать в манифесте приложения (см. главу 3) графический файл, который выводится на экран сразу после запуска, не дожидаясь инициализации JVM. Описание заставки выглядит очень просто:

```
Manifest-Version: 1.0
Main-Class: MangoMango1
SplashScreen-Image: ripe_mango.png
```

Для вывода заставки не нужно никакого кода. Изображение **ripe_mango.png** будет отображаться в центре экрана, пока не появится первое окно приложения MangoMango1 (неважно, с использованием какой графической библиотеки оно написано). Поддерживаются форматы GIF, JPEG и PNG.

Другие методы для управления окнами

Чтобы указать позицию, которую будут занимать на экране объекты `JFrame` и `JWindow`, можно воспользоваться методом `setLocation()` из класса `Component`. Координаты `x` и `y` задаются относительно левого верхнего угла экрана.

Методы `toFront()` и `toBack()` позволяют выводить объекты `JFrame` и `JWindow` над или под другими окнами. По умолчанию у пользователя есть возможность изменять размер компонента `JFrame`, но вы можете запретить это делать, если перед отображением окна вызовете метод `setResizable(false)`.

В большинстве систем окна можно «сворачивать» в небольшой значок. Для получения и задания изображения, которое будет использоваться в этом значке, предусмотрены методы `getIconImage()` и `setIconImage()`. Для окна, как и для любого другого компонента, можно изменить внешний вид указателя мыши, воспользовавшись методом `setCursor()`.

Панели

Окна имеют более сложную структуру в сравнении с обычными контейнерами. В частности, чтобы поддерживать некоторые современные возможности пользовательского интерфейса, требующие наложения компонентов друг на друга (например, контекстные меню), объекты `JFrame` и `JWindow` на самом деле состоят из отдельных перекрывающихся контейнеров (как настоящее стеклянное окно имеет несколько створок). Такие контейнеры называются «панелями» и могут быть представлены объектами нескольких разных типов: `JRootPane`, `JLayeredPane` и `GlassPane`. У объектов `JFrame` и `JWindow` всегда есть главная панель,

вмещающая в себя все содержимое окна. Это обычный контейнер, в который следует добавлять дочерние компоненты.

В целях удобства классы `JFrame` и `JWindow` делегируют вызовы методов `add()` и `setLayout()` своим главным панелям. То есть, вызовы `myFrame.add(component)` и `myFrame.getContentPane().add(component)` эквивалентны.

```
//файл: MangoMango1.java
import java.awt.*;
import javax.swing.*;
public class MangoMango1 {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Объект JFrame");
      // Следующие три вызова передаются в главную панель окна.
      frame.setLayout(new FlowLayout());
      frame.add(new JLabel("Манго"));
      frame.add(new JButton("Манго"));
      frame.setLocation(100, 100);
      frame.pack();
      frame.setVisible(true);
   }
}
```

Вызов метода `pack()` из объекта `JFrame` заставляет окно принять минимальный размер, необходимый для размещения всех дочерних компонентов. Вместо того чтобы определять точное значение, метод `pack()` говорит окну буквально следующее: «стань *достаточно* большим». Чтобы установить абсолютные размеры вручную, используйте метод `setSize()`.

Кнопки и метки мы рассмотрим уже в следующей главе, а в главе 19 речь пойдет о диспетчерах компоновки.

Интеграция с настольной средой

Одним из главных достижений платформы Java 6 стала более тесная интеграция с настольной средой, благодаря чему приложения, написанные с использованием библиотеки Swing, мало чем отличаются от системных. У разработчика появился доступ к области уведомлений, почтовому клиенту и информации о связи между типами файлов и программами.

Класс `Desktop` из пакета `java.awt` позволяет делать следующее:

- Переходить по заданному идентификатору URI с помощью стандартного браузера
- Запускать почтовый клиент, выбранный по умолчанию, передавая ему адрес получателя
- Открывать, редактировать и распечатывать файлы, используя подходящие приложения

Класс `Desktop` имеет очень простой API-интерфейс. Программа, представленная ниже, открывает стандартный веб-браузер и переходит на веб-сайт, посвященный пандам:

```
//файл: DisplayPanda.java
import java.awt.*;
import java.io.*;
import java.net.*;
public class DisplayPanda {
   public static void main(String[] args) {
      URI uri = null;
      try {
         uri = new URI("http://pandafriends.ru");
         Desktop.getDesktop().browse(uri);
      } catch(IOException ioe) {
         System.out.println("Не удалось открыть " + uri);
      } catch(URISyntaxException use) {
         System.out.println("Некорректный идентификатор URI: " + uri);
      }
   }
}
```

Все вышеупомянутые возможности реализованы в виде отдельных методов класса `Desktop` (который, к слову, используется в качестве синглтона): `open(File file)`, `edit(File file)`, `print(File file)` и `mail(URI mailtoURI)`.

Класс `SystemTray` тоже находится в пакете `java.awt` и предоставляет доступ к системной области уведомлений, где с помощью команд меню управлять запущенным приложением. В операционной системе Windows эта область находится на панели задач, а в OS X ее можно найти в правой части главной строки меню. В настольной среде GNOME она так и называется — «область уведомлений» (Notification Area).

В следующем примере мы создаем значок TrayIcon, помещаем его в область уведомлений SystemTray и привязываем к нему одну единственную команду меню, выбор которой приводит к появлению диалогового окна с приветствием.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;
public class AlohaTray {
   public static void main(String[] args) throws AWTException {
      MenuItem greetItem = new MenuItem("Приветствуйте меня");
      // Отслеживаем выбор пункта меню и выводим
      // диалоговое окно с приветствием
      greetItem.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null, "Алоха!");
            System.exit(0);
         }
      });
      // Создаем в области уведомления меню PopupMenu
      // и добавляем в него пункт MenuItem
      PopupMenu popup = new PopupMenu();
      popup.add(greetItem);
      // Создаем значок TrayIcon и добавляем его в
      // область уведомлений
      TrayIcon trayIcon = new TrayIcon(getIconImage(),
         "Дружественное приветствие", popup);
      SystemTray.getSystemTray().add(trayIcon);
   }
   // Берем для области уведомлений стандартный значок
   // из библиотеки Swing
   private static Image getIconImage() {
      Icon icon = UIManager.getIcon("OptionPane.informationIcon");
      BufferedImage image = new BufferedImage(icon.getIconWidth(),
         icon.getIconHeight(), BufferedImage.TYPE_INT_ARGB);
      icon.paintIcon(null, image.getGraphics(), 0, 0);
      return image;
   }
}
```

События

Мы потратили много времени на рассмотрение разных объектов библиотеки Swing, включая компоненты, контейнеры и окна (которые сами являются специальной разновидностью контейнеров). Теперь мы подробно обсудим взаимодействие между этими объектами.

Общение между разными компонентами в библиотеке Swing происходит посредством обмена событиями. То, как мы говорим о событиях, используя слова вроде «сгенерировать» или «обработать», может создать у вас впечатление, что речь идет об особенностях самого языка. Но это не так. Событие — это обычный объект, который доставляется получателю с помощью вызова обычного метода. Все остальное является чистой воды условностью. Событийная модель платформы Java представляет собой набор правил, которые определяют типы объектов, предназначенных для доставки; благодаря этим правилам программист знает, когда, как и кому нужно передавать события.

Источником событий служит один объект, зато получателей (слушателей) может быть сколько угодно. Слушатель реализует стандартные методы-обработчики, после чего регистрируется в источнике, подписываясь на определенный тип событий. Иногда между источником и слушателем может находиться специальный адаптер, но в любом случае доставка событий происходит только после выполнения регистрации.

Экземпляр события является производной класса `java.util.EventObject`; он хранит информацию о том, что произошло с его источником. Сам по себе класс `EventObject` служит преимущественно для идентификации объектов событий; все, что он содержит — это ссылка на источник (объект, который отправляет события). Компоненты, как правило, обмениваются не объектами `EventObject`, а их производными, которые предоставляют больше конкретики.

У класса `java.awt.EventObject` есть наследник, `AWTEvent`, дальнейшие производные которого передают информацию об определенных типах событий. В библиотеке Swing события наследуются напрямую от `EventObject`. В большинстве случаев вы будете иметь дело именно с этими дочерними классами из пакетов Swing и AWT.

События типа `ActionEvent` связаны с «действиями», которые пользователь выполняет по отношению к компоненту; это может быть щелчок по кнопке или нажатие клавиши **Enter**. Внутри события хранится назва-

ние действия (команда), которое выполняется программой. Например, события типа MouseEvent генерируются во время использования мыши, когда указатель находится над областью компонента. Вся информация, которую описывают эти объекты (координаты указателя по осям x и y, состояние клавиш мыши), фиксируется в момент их создания.

Объекты ActionEvent работают на более высоком уровне, нежели их производные. Они дают нам знать, что компонент выполнил свою задачу, тогда как, например, экземпляр MouseEvent просто передает нам некий срез информации о состоянии мыши в заданный момент времени. Конечно, определить нажатие кнопки JButton можно и с помощью объекта MouseEvent, но событие типа ActionEvent позволяет сделать это проще. Также стоит помнить, что конкретный смысл события может зависеть от контекста, в котором оно было получено.

Получатели событий и интерфейс EventListener

События передаются в виде аргументов для методов-обработчиков, принадлежащих объекту-получателю. Например, экземпляры ActionEvent всегда доставляются посредством метода actionPerformed():

```
public void actionPerformed( ActionEvent e ) {
   ...
}
```

Интерфейс слушателя должен предоставлять соответствующие методы для тех типов событий, которые он хочет получать. В данном случае любой объект, принимающий события типа ActionEvent, обязан реализовать интерфейс ActionListener:

```
public interface ActionListener extends java.util.EventListener {
   public void actionPerformed( ActionEvent e );
}
```

Все получатели событий должны наследовать пустой интерфейс java.util.EventListener. Это нужно только для того, чтобы различным инструментам разработки, таким как IDE, было проще идентифицировать объект-слушатель.

Требование к наличию интерфейса у слушателя вызвано несколькими причинами. Во-первых, это помогает понять, может ли объект полу-

чать указанный вид событий — каждый обработчик поддерживает строго определенный тип. Сами методы, предназначенные для обработки событий, могут иметь понятные, наглядные имена, и при этом хорошо подходить для документирования и автоматического (или ручного) распознавания среди других членов класса. Во-вторых, объект может реализовывать сразу несколько методов интерфейса слушателя. Например, интерфейс `FocusListener` содержит два обработчика:

```
abstract void focusGained( FocusEvent e );
abstract void focusLost( FocusEvent e );
```

И хотя оба эти метода принимают аргументы типа `FocusEvent`, причины, по которым они вызываются (контекст), носят разный характер: первый метод сообщает о выделении элемента, а второй говорит о потере фокуса. В этом случае настоящую причину можно также узнать по содержимому полученного аргумента; все производные класса `AWTEvent` хранят константу, в которой указывается тип события. Чтобы обойтись без этих проверок, можно просто использовать два разных обработчика в рамках интерфейса `FocusListener` — например, вызов метода `focusGained()` сам по себе означает, что событие имеет тип `FOCUS_GAINED`.

Похожим образом дела обстоят и с интерфейсом `MouseListener`. В нем объявлено пять методов для получения событий, связанных с работой мыши (в интерфейсе `MouseMotionListener` есть еще два), каждый из которых предоставляет дополнительный контекст. Обычно в интерфейсе слушателя собраны обработчики, которые как-то между собой связаны; если метод вызывается в любой ситуации, информацию о контексте стоит искать в самом экземпляре события.

Бывают типы событий, для которых предусмотрено сразу несколько интерфейсов слушателей. Например, интерфейс `MouseListener` описывает методы для обработки событий `MouseEvent`; они возникают при нажатии клавиш мыши, а также в моменты, когда указатель попадает в область элемента и выходит из нее. Но есть совершенно другой интерфейс, `MouseMotionListener`, методы которого получают события того же типа, когда указатель мыши двигается или выполняет перетаскивание (движение с нажатой клавишей) над элементом. Разделяя объекты `MouseEvent` на две категории, платформа Java позволяет вам учитывать обстоятельства, при которых вы хотите принимать события данного типа. Например, вы можете зарегистрировать объект-слушатель так, чтобы он следил за активностью мыши, но не получал при этом огром-

ный объем лишней информации о перемещениях указателя, если она вас не интересует.

В библиотеке Swing объявления интерфейсов слушателей и их методов-обработчиков выполнены с учетом двух требований:

- Обработчики событий являются публичными методами, используют **void** в качестве типа возвращаемого значения и принимают только один аргумент — производную класса `java.util.EventObject`.*
- Слушатель наследует интерфейс `java.util.EventListener`, а его имя должно иметь суффикс «`Listener`» (например, `MouseListener` или `ActionListener`).

Данные правила могут показаться очевидными, но от этого они не становятся менее важными. Это основные рекомендации, которыми следует руководствоваться при построении компонентов с поддержкой событий.

Источники событий

В предыдущем разделе описывался механизм, с помощью которого получатель отслеживает события. Теперь мы поговорим о том, как источник событий узнает, с какими получателями ему нужно иметь дело.

Чтобы получать события, слушатель должен быть зарегистрирован в источнике, который их генерирует, и иметь методы для их обработки. Для выполнения регистрации у источника событий предусмотрен специальный метод, который принимает ссылку на объект-слушатель (в итоге получается *функция обратного вызова*). Например, класс `JButton` является источником событий типа `ActionEvent`. Чтобы принимать эти события, объект `TheReceiver` должен зарегистрироваться следующим образом:

```
// получатель событий ActionEvent
class TheReceiver implements ActionListener
{
   // источник событий ActionEvent
   JButton theButton = new JButton("Пупок");
```

* Это правило не является обязательным. Спецификация технологии JavaBeans (см. главу 22) позволяет обработчикам принимать дополнительные аргументы и генерировать проверяемые исключения, если без этого никак нельзя обойтись.

```
TheReceiver() {
    ...
    theButton.addActionListener( this );
}
public void actionPerformed( ActionEvent e ) {
    // Кнопка с надписью "Пупок" была нажата...
}
```

Чтобы подписаться на получение событий типа `ActionEvent`, объект `TheReciever` вызывает из кнопки метод `addActionListener()`. Регистрируясь, он передает ссылку **this**, потому как сам является реализацией интерфейса `ActionListener`.

Для управления своими слушателями источники событий типа `ActionEvent` (такие как `JButton`) всегда реализовывают два метода:

```
// источник событий ActionEvent
public void addActionListener(ActionListener listener) {
    ...
}
public void removeActionListener(ActionListener listener) {
    ...
}
```

Метод `removeActionListener()` удаляет объект из списка слушателей, после чего тот уже не сможет получать события данного типа. Компоненты библиотеки Swing содержат оба метода; обычно их не нужно реализовывать вручную. Важно уделять внимание тому, каким образом ваше приложение использует источники событий и объекты-слушатели. Удаление источника без отмены регистрации его слушателей — вполне нормальная практика, но если вы удалите слушателя, который все еще зарегистрирован, ссылка не него может остаться в источнике; в этом случае он не будет утилизирован сборщиком мусора.

Вы, наверное, думаете, что эти два метода определены в специальном интерфейсе, который идентифицирует источник событий определенного типа. Но это не так. В соответствии с текущими соглашениями у источника нет своего интерфейса. Чтобы понять, какие события генерирует класс, вам нужно обратить внимание на его методы с префиксами «add» и «remove». К примеру, наличие методов `addActionListener()` и `removeActionListener()` будет говорить о том, что класс является источником событий типа `ActionEvent`. Это поможет вам разобраться в назначении компонента, если под рукой не окажется документации, или

если вы пишете программу, которая должна анализировать исходный код (этот процесс называется «*отражением*»). Во втором случае вам может пригодиться класс `java.beans.Introspector`.

Итак, источник, который генерирует для интерфейса `FooListener` события типа `FooEvent`, должен предоставлять два метода:

- `addFooListener(FooListener` *`listener`* `)`
- `removeFooListener(FooListener` *`listener`* `)`

Если источник событий поддерживает только одного слушателя (однонаправленная передача), метод с префиксом «add» может сгенерировать исключение `java.util.TooManyListenersException`.

Что мы получаем, используя эти соглашения об именовании? С одной стороны, это позволяет находить источники определенных событий с помощью автоматических инструментов и интегрированных сред разработки. Инструменты для работы с технологией JavaBeans будут искать подобного рода методы с помощью таких приемов, как отражение и интроспекция, пытаясь идентифицировать события, которые могут быть сгенерированы компонентом.

На практике это означает, что обработчики событий строго типизированы, как и любая другая часть платформы Java. Благодаря этому вы не можете ошибиться при подключении к компонентам; например, у вас не получится подписаться на отслеживание событий типа `ItemEvent`, генерируемых кнопкой, так как в классе `JButton` нет метода `addItemListener()`. Компилятор знает, кому позволено получать те или иные виды событий.

Доставка событий

Библиотеки Swing и AWT поддерживают многоадресную доставку; каждое событие связано только с одним источником, но получателей у него может быть сколько угодно. При срабатывании событие доставляется отдельно каждому слушателю в списке (см. рис. 16.3).

Порядок, в котором события будут доставлены, нельзя предугадать заранее. Также невозможно предвидеть последствия многократной регистрации в одном и том же источнике — вам может прийти только одно событие, а может сразу несколько. По этой же причине вам нужно исходить из того, что каждый слушатель получает одни и те же данные, так как события нельзя изменить после получения.

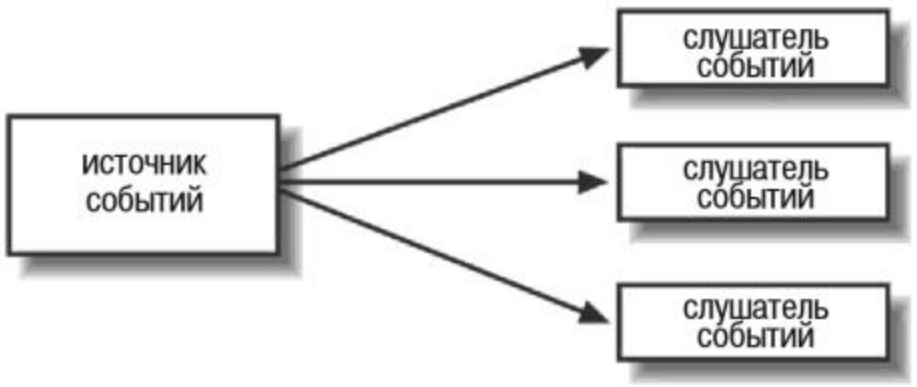

Рис. 16.3. Доставка событий

В дополнение к вышесказанному мы могли бы отметить, что доставка событий является синхронной по отношению к источнику, но это вызвано лишь тем фактом, что данный механизм основан на вызове обычных методов. Источник событий просто вызывает обработчик каждого слушателя. Тем не менее слушатели должны учитывать, что события могут передаваться в разных потоках выполнения; исключения составляют библиотеки Swing и AWT, которые всегда отправляют свои события последовательно, используя для этого специальный глобальный поток.

Типы событий

Все события, которые применяются в графических компонентах библиотеки Swing, являются производными класса `java.util.EventObject`. В своих компонентах вы можете использовать или наследовать любые разновидности этого типа. В этом разделе мы опишем наиболее важные виды событий.

В библиотеке Swing события и слушатели содержатся в двух пакетах: `java.awt.event` и `javax.swing.event`. Как уже было сказано, компоненты, предоставляемые библиотеками Swing и AWT, заметно отличаются по своей структуре, хотя их событийные модели работают по одному и тому же принципу. Таким образом, элементы управления библиотеки Swing совместимы с событиями и слушателями из обоих пакетов, `java.awt.event` и `javax.swing.event`.

Класс `java.awt.event.ComponentEvent` является базовым для событий, которые могут быть сгенерированы любым компонентом. Это касается ситуаций, связанных с изменением свойств элементов управления (таких как видимость или размеры), операциями с мышью или нажатиями клавиш на клавиатуре. В качестве источника событий типа `ContainerEvent` может также выступить контейнер, который генерирует их при добавлении и удалении компонентов.

Класс `java.awt.event.InputEvent`

События `MouseEvent`, позволяющие отслеживать состояние мыши, и `KeyEvent`, которые срабатывают при использовании клавиатуры, являются производными класса `java.awt.event.InputEvent`. Они генерируются, когда пользователь нажимает клавишу или перемещает указатель мыши; в качестве их источника выступает компонент, в области которого выполняются эти действия.

События, связанные с вводом и пользовательским интерфейсом, обрабатываются в специальной очереди. Это позволяет библиотеке Swing управлять доставкой каждого своего события. В некоторых ситуациях из нескольких последовательных событий одного типа до получателя может дойти только одно. Это сделано для того, чтобы повысить эффективность некоторых аспектов библиотеки, таких как работа с мышью и обмен внутренними сообщениями для перерисовки компонентов. Но для нас, вероятно, более важным является тот факт, что события ввода содержат дополнительную информацию, исходя из которой слушатели могут решать, должны ли компоненты сами выполнять обработку.

Модификаторы, связанные с мышью и клавиатурой

События `InputEvent` содержат набор флагов, которые соответствуют клавиатурным модификаторам. Они позволяют определить, какой кнопкой мыши был выполнен щелчок и были ли при этом (или при вводе с клавиатуры) нажаты клавиши **Shift**, **Ctrl** или **Alt**. Класс `java.awt.event.InputEvent` предоставляет следующие флаги:

`SHIFT_MASK`

Во время события была нажата клавиша **Shift**.

`CTRL_MASK`

Во время события была нажата клавиша **Ctrl**.

`ALT_MASK`

Во время события были нажаты клавиша **Alt** (Windows) или **Option**/**Alt** (OS X); эквивалентно флагу `BUTTON2_MASK`.

`META_MASK`

Во время события была нажата клавиша **Command** (OS X); эквивалентно флагу `BUTTON3_MASK`.

`BUTTON1_MASK`

Левая кнопка мыши.

`BUTTON2_MASK`

Средняя кнопка мыши; эквивалентно флагу `ALT_MASK`.

`BUTTON3_MASK`

Правая кнопка мыши; эквивалентно флагу `META_MASK`.

Чтобы проверить один или несколько флагов, укажите перед ними полный набор модификаторов в сочетании с побитовой операцией И. Получить все модификаторы, связанные с событием, можно с помощью метода `getModifiers()` из объекта `InputEvent`:

```
public void mousePressed (MouseEvent e) {
   int mods = e.getModifiers();
   if ((mods & InputEvent.SHIFT_MASK) != 0) {
      // щелчок мышью с нажатой клавишей Shift
   }
}
```

Три флага с префиксом «BUTTON» позволяют определить, какая из клавиш мыши была нажата. Флаги `BUTTON2_MASK` и `ALT_MASK` эквивалентны, так же, как `BUTTON3_MASK` и `META_MASK`. Другими словами, щелчок левой клавиши мыши в сочетании с нажатыми клавишами **Alt** или **Meta** имеет такой же эффект, как нажатие средней или правой клавиш соответственно. Это обеспечивают некую минимальную совместимость даже с теми компьютерами, у которых нет многоклавишной мыши. Однако самая распространенная операция, которая выполняется с помощью этих клавиш — вызов контекстного меню — не требует от вас написания отдельного кода; она предоставляется библиотекой Swing, и в каждой системе имеет свою реализацию (см. класс `PopupMenu` в главе 17).

События, связанные с колесом мыши

В платформе Java 1.4 появилась поддержка колеса мыши — устройства для прокрутки, которое находится на месте средней клавиши. Библиотека Swing сама обрабатывает вращение колеса для компонентов, которые

поддерживают прокрутку, поэтому вам не нужно реализовывать это вручную. События, связанные с колесом мыши, обрабатываются особым образом, так как указатель не всегда обязан находиться над компонентом, который прокручивается. Если текущий элемент управления не подписан на получение этих событий, библиотека Swing начинает перебирать его контейнеры в поисках подходящего обработчика. Чтобы компонент вел себя предсказуемо, его можно поместить в контейнер типа `ScrollPane`.

Если вы хотите самостоятельно обрабатывать вращение колеса мыши, можете зарегистрироваться для получения соответствующих событий с помощью интерфейса `MouseWheelListener` (см. табл. 16.1 в следующем разделе). Эти события инкапсулируют информацию о протяженности прокручивания и единицах вращения (которые в большинстве систем пользователь может задать отдельно). Чтобы узнать физические параметры вращения, можете воспользоваться методом `getWheelRotation()`; он возвращает количество делений, на которые было повернуто колесо.

События, связанные с выделением компонентов

Как уже упоминалось ранее, обработка выделения компонентов в библиотеке Swing выполняется преимущественно автоматически, и мы еще вернемся к этому вопросу в главе 18. Но понимание того, как работают события, связанные с фокусировкой, поможет вам лучше ориентироваться в элементах управления и даст вам возможность изменять их под свои нужды.

Вы уже знаете, что сделать компонент доступным для выделения можно с помощью метода `setFocusable()` (в операционной системе Windows может применяться его аналог, `setFocusableWindowState()`). Обычно компонент получает фокус, когда пользователь делает по нему щелчок мышью. Это можно сделать и программным способом, воспользовавшись методами `requestFocus()` или `requestFocusInWindow()`; они делают практически одно и то же, только второй не запрашивает перемещение фокуса между окнами. (На сегодняшний день на некоторых платформах существуют ограничения, которые не дают передавать выделение из «родной» программы в приложение, написанное на языке Java; метод `requestFocusInWindow()` учитывает это обстоятельство, обеспечивая переносимость кода.)

И хотя компонент может запросить выделение явным образом, получение или потерю фокуса можно проверить исключительно с помощью интерфейса `FocusListener` (см. табл. 16.1 и 16.2). Этот интерфейс позво-

ляет изменять поведение компонента в момент, когда тот готов для ввода (например, когда мигает текстовый курсор в компоненте `TextField`). Кроме того, некоторые компоненты обрабатывают введенные данные в момент потери фокуса. Например, когда пользователь переходит между полями ввода типа `JTextField`, программа может проверять текст, который уже был набран; это позволит предотвратить смену выделения в случае ввода некорректной информации (в главе 18 вы увидите, как это работает).

В момент выделения компонента происходит следующая последовательность событий (при условии, что компонент не был выделен до этого):

```
WINDOW_ACTIVATED
WINDOW_GAINED_FOCUS
FOCUS_GAINED
```

Первые два события имеют тип `WindowEvent` и доставляются в контейнер `Window`, в котором содержится компонент; третье, `FocusEvent`, передаются непосредственно компоненту. Вот что произойдет, если выделить компонент в другом окне:

```
FOCUS_LOST
WINDOW_FOCUS_LOST
WINDOW_DEACTIVATED
```

Эти события несут в себе определенный контекст, который помогает принимающему объекту определить, откуда был передан фокус. Компонент и окно, которые были выделены до этого, называются «обратными»; доступ к ним можно получить с помощью методов `FocusEvent.getOppositeComponent()` и `WindowEvent.getOppositeWindow()`. Возвращаемые ими значения могут равняться `null`, если приложение, от окна которого перешел фокус, написано не на языке Java.

События, связанные с получением и потерей фокуса, могут быть помечены как «временные»; определить это можно с помощью метода `isTemporary()` из объекта `FocusEvent`. Концепция временного изменения фокуса применяется в таких элементах управления, как контекстные меню, полосы прокрутки и заголовки окон, которые после своего использования возвращают управление предыдущему компоненту. Временная потеря фокуса отличается от обычной тем, что она не должна приводить к проверке или дальнейшему использованию введенной информации.

Список часто используемых событий

В табл. 16.1 и 16.2 собраны события, которые часто используются в библиотеке Swing. Здесь также приводятся компоненты, которые их генерируют, и методы интерфейсов слушателей, которые их получают. Все записи сгруппированы по пакетам `java.awt.event` и `javax.swing.event`.

Табл. 16.1. События, генерируемые компонентами и контейнерами библиотеки Swing

Событие	Источники	Интерфейс слушателя	Обработчики

`java.awt.event.ComponentEvent`	Все компоненты	`ComponentListener`	`componentResized()`
			`componentMoved()`
			`componentShown()`
			`componentHidden()`

`java.awt.event.FocusEvent`	Все компоненты	`FocusListener`	`focusGained()` `focusLost()`

`java.awt.event.KeyEvent`	Все компоненты	`KeyListener`	`keyTyped()`
			`keyPressed()`
			`keyReleased()`

`java.awt.event.MouseEvent`	Все компоненты	`MouseListener`
		
		`MouseMotionListener`
			`mouseClicked()`
			`mousePressed()`
			`mouseReleased()`
			`mouseEntered()`
			`mouseExited()`
			`mouseDragged()`
			`mouseMoved()`

`java.awt.event.ContainerEvent`	Все компоненты	`ContainerListener`	`componentAdded()` `componentRemoved()`

Табл. 16.2. События, связанные с определенными компонентами

Событие	Источники	Интерфейс слушателя	Обработчики

`java.awt.event.ActionEvent`	`JButton`
	`JCheckBoxMenuItem`
	`JComboBox`
	`JFileChooser`
	`JList`
	`JRadioButton`
	`MenuItem`
	`JTextField`
	`JToggleButton`
		`ActionListener`	`actionPerformed()`

`java.awt.event.AdjustmentEvent`	`JScrollBar`	`Adjustment Listener`	`adjustmentValueChanged()`
`javax.swing.event.CaretEvent`	`JTextComponent`	`CaretListener`	`caretUpdate()`

Событие	Источники	Интерфейс слушателя	Обработчики
javax.swing.event.HyperlinkEvent	JEditorPane JTextPane	HyperlinkListener	hyperlinkUpdate()
java.awt.event.InternalFrameEvent	JInternalFrame	InternalFrameListener	internalFrameActivated() internalFrame Closed() internalFrameClosing() internalFrameDeactivated() internalFrameDeiconified() internalFrameIconified() internalFrame Opened()
java.awt.event.ItemEvent	JCheckBox MenuItem JComboBox JRadioButton MenuItem JToggleButton	ItemListener	itemStateChanged()
javax.swing.event.ListDataEvent	ListModel	ListDataListener	contentsChanged() intervalAdded() intervalRemoved()
javax.swing.event.ListSelectionEvent	JList ListSelectionModel	ListSelectionListener	valueChanged()
javax.swing.event.MenuEvent	JMenu	MenuListener	menuCanceled() menuDeselected() menuSelected()
javax.swing.event.PopupMenuEvent	JPopupMenu	PopupMenuListener	popupMenuCanceled() popupMenuWillBecomeInvisible() popupMenuWillBecomeVisible()
javax.swing.event.MenuKeyEvent	JMenuItem	MenuKeyListener	menuKeyPressed() menuKeyReleased() menuKeyTyped()
javax.swing.event.MenuDragMouseEvent	JMenuItem	MenuDragMouseListener	menuDragMouseDragged() menuDragMouseEntered() menuDragMouseExited() menuDragMouseReleased()
javax.swing.event.TableColumnModelEvent	TableColumnModel*	TableColumnModelListener	columnAdded() columnMargin Changed() columnMoved() columnRemoved() columnSelectionChanged()
javax.swing.event.TableModelEvent	TableModel	TableModelListener	tableChanged()

Событие	Источники	Интерфейс слушателя	Обработчики
javax.swing.event.TreeExpansionEvent	Jtree	TreeExpansionListener	treeCollapsed() treeExpanded()
javax.swing.event.TreeModelEvent	TreeModel	TreeModelListener	treeNodesChanged() treeNodesInserted() treeNodesRemoved() treeStructureChanged()
javax.swing.event.TreeSelectionEvent	JTree TreeSelectionModel	TreeSelectionListener	valueChanged()
javax.swing.event.UndoableEditEvent	javax.swing.text.Document	UndoableEditListener	undoableEditHappened()
java.awt.event.WindowEvent	JDialog JFrame JWindow	WindowListener	windowOpened() windowClosing() windowClosed() windowIconified() windowDeiconified() windowActivated() windowDeactivated()

* Класс `TableColumnModel` нарушает соглашение об именовании методов для регистрации слушателей. У него они называются `addColumnModelListener()` и `removeColumnModelListener()`.

В компонентах библиотеки Swing модель и представление отделены друг от друга. Строго говоря, события генерирует не сам компонент, а его модель данных. Например, когда вы нажимаете кнопку `JButton`, событие `ActionEvent` передается вам из модели, а не из кнопки как таковой. Но у объекта `JButton` есть удобный метод для регистрации слушателей типа `ActionListener`; при вызове он передает аргумент своей модели, которая и выполняет регистрацию. Обычно (как в случае с `JButton`) программисту не нужно разделять работу с моделью и представлением, поэтому мы сознательно допускаем некоторую неточность, когда говорим, что компонент сам генерирует события. Конечно, мы понимаем, что объекты типа `InputEvent` на самом деле создаются стандартной системой ввода, но слушатели реагируют на них так, как будто они были сгенерированы соответствующим элементом управления.

Классы адаптеров

Ситуация, в которой компоненты приложения реализуют множество интерфейсов слушателей, чтобы получать события напрямую, является далеко не идеальной. Иногда она просто невозможна. Чтобы

реализовать нужный интерфейс и добавить код для обработки событий, придется модифицировать или наследовать класс получателя. А поскольку речь идет о событиях из библиотеки Swing, перед нами встает еще одна, менее очевидная проблема: код, отвечающий за отображение компонентов, может попасть в те части вашего приложения, которые не должны ничего знать о пользовательском интерфейсе. Давайте рассмотрим пример.

На рис. 16.4 изображена структура приложения для управления кухонным комбайном. Наш класс `Vegomatic` реализует интерфейс `ActionListener`, чтобы принимать события непосредственно от трех кнопок: `chopButton` (измельчает), `pureeButton` (делает пюре) и `frappeButton` (готовит фруктовый напиток со льдом).

Проблема в том, что теперь объекту `Vegomatic` нужно знать о том, как обрабатывать продукты. К тому же ему теперь известно, что он управляется тремя компонентами (кнопками, которые отправляют команды) и что в ответ на полученные события необходимо вызывать определенные методы. Из-за этого два прямоугольника, обозначающих пользовательский интерфейс и код приложения, стали пересекаться. Позже, когда парни из маркетингового отдела попробуют сделать что-то с этими кнопками (возможно, поменять их названия), нам придется изменять логику нашего объекта `Vegomatic`. Это никуда не годится.

Чтобы избежать подобных проблем, между источником и получателем событий можно разместить специальный адаптер. *Адаптер* — это обычный объект, чья единственная задача заключается в привязке поступающих событий к внешнему методу.

На рис. 16.5 показана более удачная структура приложения, в которой для каждой из трех кнопок используется свой адаптер. Реализация первого адаптера могла бы выглядеть примерно так:

```
class VegomaticAdapter1 implements ActionListener {
   Vegomatic vegomatic;
   VegomaticAdapter1 ( Vegomatic vegomatic ) {
      this.vegomatic = vegomatic;
   }
   public void actionPerformed( ActionEvent e ) {
      vegomatic.chopFood();
   }
}
```

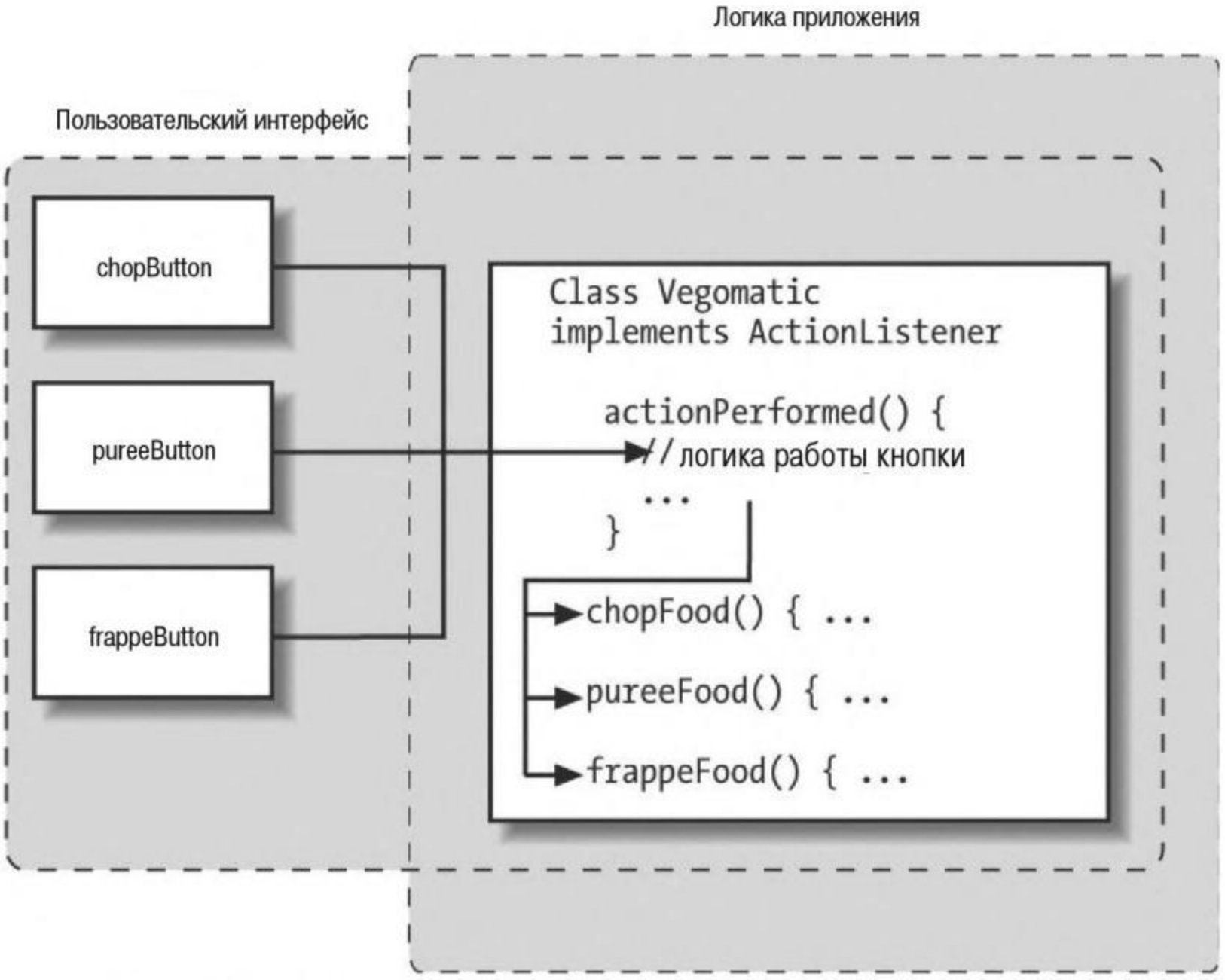

Рис. 16.4. Реализация интерфейса `ActionListener` без посредников

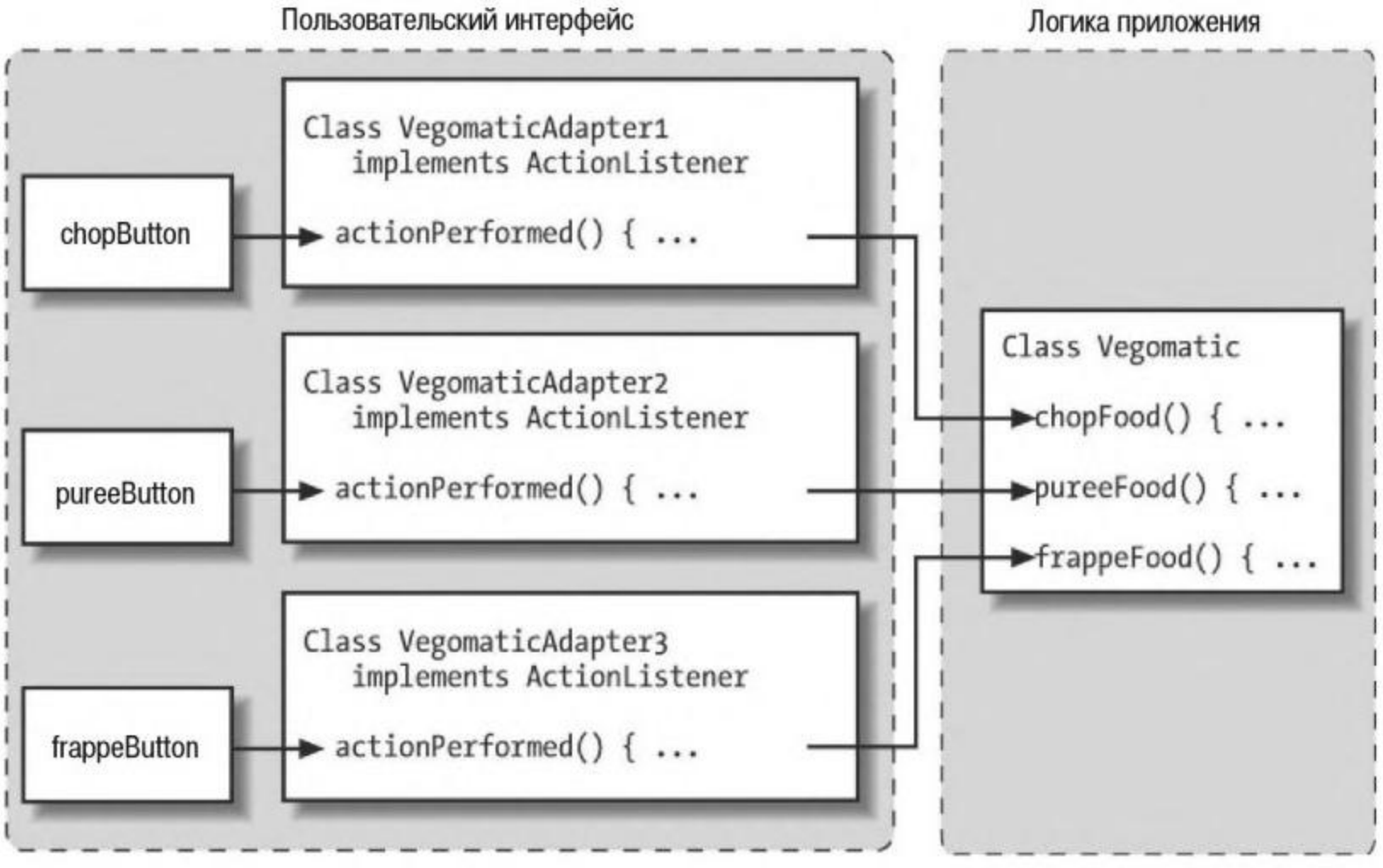

Рис. 16.5. Реализация интерфейса `ActionListener` с помощью адаптеров

Итак, в той части приложения, где формируется пользовательский интерфейс, мы могли бы зарегистрировать наш объект-слушатель следующим образом:

```
Vegomatic theVegomatic = ...;
Button chopButton = ...;
// делаем привязку
chopButton.addActionListener( new VegomaticAdapter1(theVegomatic) );
```

В качестве слушателя для кнопки `JButton` наш адаптер регистрирует не себя (`this`), а объект `Vegomatic` (`theVegomatic`). Выступая посредником, он связывает источник событий (кнопку) с получателем (виртуальным измельчителем).

Мы устранили былой беспорядок, полностью отделив пользовательский интерфейс от остального кода приложения. Хотя при этом у нас появилось три новых класса, которые, в сущности, не делают ничего полезного. Хорошо это или плохо — нельзя ответить однозначно.

В других обстоятельствах наши кнопки могли бы иметь один общий адаптер, экземплярам которого передавались бы разные параметры. В поисках разумного компромисса мы можем варьировать такие аспекты нашего кода, как размер, эффективность и изящество. Классы адаптеров часто генерируются автоматически различными инструментами; на это мы намекнули, выбрав имена `VegomaticAdapter1`, `VegomaticAdapter2` и `VegomaticAdapter3`.

При ручном создании адаптеров обычно используют анонимные внутренние классы (об этом мы поговорим в следующем разделе). Хотя мы вообще можем отказаться от строгой типизации, которую навязывает язык Java, и налаживать связь между источником и слушателем событий динамически, используя технологию Reflection API.

Шаблонные адаптеры

Многие интерфейсы слушателей содержат сразу несколько обработчиков событий, и чтобы подписаться на какое-то одно из них, вам придется реализовать весь интерфейс. В итоге ваш код будет содержать шаблонные «методы-заглушки», которые ничего не делают. В этом нет ничего плохого, но написание такого кода вручную является довольно скучным занятием. Чтобы освободить вас от этой рутины, библиотеки AWT и Swing предлагают набор вспомогательных классов, которые берут

на себя реализацию шаблонных методов. У всех наиболее распространенных интерфейсов слушателей, которые содержат более одного обработчика, есть соответствующие классы адаптеров с методами-заглушками. Вы можете использовать их как отправную точку при создании своих собственных адаптеров.

Когда возникнет необходимость установить связь между источником и слушателем событий, просто унаследуйте один из этих классов и переопределите только те методы, которые вам нужны.

Например, класс `MouseAdapter` реализует интерфейс `MouseListener`, предоставляя следующую минималистическую реализацию:

```
public void mouseClicked(MouseEvent e) {};
public void mousePressed(MouseEvent e) {};
public void mouseReleased(MouseEvent e) {};
public void mouseEntered(MouseEvent e) {};
public void mouseExited(MouseEvent e) {};
```

Это не сэкономит вам так уж много времени, но иметь такой инструмент под рукой никогда не помешает. Лучше всего реализацию `MouseAdapter` использовать в качестве основы для собственного адаптера в рамках анонимного внутреннего класса.

Представьте, что нам нужно получить из некоторого компонента событие `mousePressed()` и выполнить после этого какое-то действие. Для этого можно создать следующую привязку:

```
someComponent.addMouseListener( new MouseAdapter() {
   public void MousePressed(MouseEvent e) {
      obj.execute();
   }
} );
```

Мы позволили себе некоторые вольности при форматировании, но код все равно выглядит довольно понятным. Кроме того, нам не пришлось создавать заглушки для четырех обработчиков, которые не используются. Написание адаптеров — довольно частое занятие для программиста, поэтому, чем меньше лишнего кода, тем лучше; возможно, это поможет вам немного отсрочить проявление симптомов туннельного синдрома. Помните, что на этапе компиляции внутренний класс всегда превращается в обычный, поэтому неразбериха, которую вы замаскировали в своем исходном коде, никуда не девается.

Настоящий робот в библиотеке AWT!

Эта тема может не иметь какого-то практического значения, но иногда API-интерфейсы настолько интересны, что о них просто нельзя не упомянуть. В платформе Java 1.3 появился класс с интригующим названием `java.awt.Robot`. Этот «робот» предоставляет интерфейсы для программного создания событий ввода, таких как нажатия клавиш или движение указателя мыши. С его помощью можно разрабатывать, например, автоматизированные системы для тестирования пользовательских интерфейсов. В следующем примере класс `Robot` перемещает указатель мыши в левый верхний угол экрана и выполняет несколько двойных щелчков.

В большинстве систем семейства Windows это приводит к открытию папки *Мой компьютер*, ярлык которой находится как раз в той части рабочего стола.

```
public class RobotExample
{
   public static void main( String [] args ) throws Exception
   {
      Robot r = new Robot();
      r.mouseMove(35,35);
      r.mousePress( InputEvent.BUTTON1_MASK );
      r.mouseRelease( InputEvent.BUTTON1_MASK );
      Thread.sleep(50);
      r.mousePress( InputEvent.BUTTON1_MASK );
      r.mouseRelease( InputEvent.BUTTON1_MASK );
   }
}
```

Помимо волшебных пальцев у этого робота есть еще и глаза! Метод `createScreenCapture()` из класса `Robot` позволяет делать снимки экрана или его отдельных прямоугольных участков (точные размеры экрана можно получить с помощью метода `getScreenSize()`, который предоставляет библиотека AWT).

В платформу Java 5.0 был добавлен еще один полезный API-интерфейс, `java.awt.MouseInfo`, который позволяет собирать информацию о движении указателя мыши в любой части экрана (не ограничиваясь окном приложения). Использование классов `Robot` и `MouseInfo` должно облегчить запись и воспроизведение любых событий ввода.

Многопоточность в библиотеке Swing

На ранних этапах создания библиотеки Swing ее разработчикам пришлось пойти на некоторые компромиссы, связанные со скоростью, однородностью пользовательского интерфейса и потоковой безопасностью. Чтобы в целом добиться максимальной производительности и простоты, библиотека Swing не занимается синхронизацией доступа к методам своих компонентов. Это означает, что большинство элементов управления, которые она предоставляет, не являются безопасными для использования в многопоточных приложениях. Но не волнуйтесь: все не так плохо, как может показаться. Так и было задумано. Все события в библиотеках AWT и Swing помещаются в отдельную очередь и обрабатываются в рамках единого системного потока. Очередь выполняет сразу две функции. Во-первых, она нивелирует все проблемы, связанные с потоковой безопасностью, так как любые изменения пользовательского интерфейса происходят в однопоточном режиме. Во-вторых, она гарантирует строгое соблюдение порядка обработки событий. И так как перерисовка компонентов в библиотеке Swing основана на событийной модели, все обновления экрана привязаны к соответствующим обработчикам.

С точки зрения программиста это означает, что в многопоточных программах необходимо осторожно подходить к обновлению компонентов, которые были *реализованы* (добавлены в видимый контейнер). Компоненты, которые модифицируются из внешних потоков, могут выводиться в искаженном виде и вести себя некорректно.

Есть несколько условий, соблюдение которых гарантирует безопасность изменения компонентов библиотеки Swing. Во-первых, изменять компоненты можно перед их реализацией. Термин «реализация» начали использовать в те времена, когда компонент должен был создавать для себя связующий объект. Это этап, на котором он добавляется в видимый контейнер (или просто становится видимым, если он размещен непосредственно в окне). В большинстве примеров, которые мы приводим в этой книге, компоненты подготавливаются внутри метода `main()`; там они добавляются в контейнер `JFrame`, который в итоге выводится на экран с помощью метода `setVisible()`. Такой подход является безопасным, так как элементы управления реализуются только после того, как их контейнер становится видимым. Хотя, строго говоря, это не совсем так. Формально компонент можно реализовать с помощью метода `pack()` из объекта `JFrame`. Но вероятность того, что пользовательский

интерфейс начнет работать некорректно еще до своего появления на экране, крайне мала.

Во-вторых, безопасное изменение можно проводить с помощью кода, который сам выполняется в системном потоке обработки событий. Так как события находятся в очереди, все их обработчики обычно вызываются именно из этого потока. Это означает, что любые методы слушателя и, соответственно, вызовы, которые они делают на протяжении своего жизненного цикла, могут свободно изменять элементы пользовательского интерфейса, потому как вся работа происходит в одном и том же потоке выполнения. Вы можете идентифицировать текущий поток с помощью статического метода `SwingUtilities.isEventDispatchThread()`, чтобы в случае необходимости вернуться в рамки очереди событий (которую мы обсудим далее) и исключить возможность вызова вашего кода извне.

Кроме того, модифицировать компоненты можно в том случае, если в документации к API-интерфейсу прямо сказано, что конкретная операция является потокобезопасной. Такие методы встречаются во многих компонентах библиотеки Swing — например, в классе `JComponent` (методы `repaint()` и `revalidate()`) или в элементах управления для работы с текстом. Это также касается методов добавления и удаления во всех интерфейсах слушателей.

Если ни одно из вышеперечисленных условий потоковой безопасности вам не подходит, вы можете сами получить доступ к системной очереди событий и выполнить свой код в рамках единого потока. На этот случай предусмотрено два статических метода из класса `javax.swing.SwingUtilities`: `invokeAndWait()` и `invokeLater()`.

```
public static void invokeLater(Runnable doRun)
```

Позволяет вызвать метод `run()` из специальной реализации `Runnable`.

```
public static void invokeAndWait(Runnable doRun) throws
InterruptedException, InvocationTargetException
```

Делает то же самое, что `invokeLater()`, но дожидается завершения работы метода `run()`.

Любые операции, которые вы делегировали объекту `Runnable`, будут выполнены этими методами в рамках механизма передачи событий. Часто для этого используются внутренние классы. Например:

```
SwingUtilities.invokeLater( new Runnable() {
   public void run() {
      MyComponent.setVisible(false);
   }
} );
```

В платформе Java 7 был представлен новый класс под названием `SwingWorker`; он помогает в ситуациях, когда вашему фоновому процессу после завершения или во время работы нужно обновлять пользовательский интерфейс. В первом случае весь долгоиграющий код лучше поместить в метод `doInBackground()`, а обновление выполнять в методе `done()`.

```
package learning;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
public class MysteryOfTheUniverse extends JFrame {
   JTextArea textArea;
   JButton solveButton;
   public MysteryOfTheUniverse() {
      super("Разгадываем тайны мироздания");
      setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      setSize(300, 300);
      textArea = new JTextArea();
      solveButton = new JButton("Разгадать загадку");
      solveButton.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            solveButton.setEnabled(false);
            solveMysteryOfTheUniverse();
         }
      });
      add(solveButton, BorderLayout.NORTH);
      add(new JScrollPane(textArea));
      add(new JButton("Нажми меня! Я неблокирующая."),
         BorderLayout.SOUTH);
   }
   public void solveMysteryOfTheUniverse() {
      (new MysteryWorker()).execute();
   }
   class MysteryWorker extends SwingWorker<String, Object> {
      @Override
```

```
    public String doInBackground() {
        // Думаем 4 секунды, не блокируя интерфейс
        try {
            Thread.currentThread().sleep(4000);
        } catch (InterruptedException ignore) {}
        solveButton.setEnabled(true);
        return "Яичный салат";
    }
    @Override
    protected void done() {
        try {
            textArea.setText(get());
        } catch (Exception ignore) {}
      }
    }
    public static void main(String[] args) {
        new MysteryOfTheUniverse().setVisible(true);
    }
}
```

В результате нажатия кнопки `solveButton` приложение потратит четыре секунды на разгадку тайны мироздания. Если бы мы не использовали объект `SwingWorker`, поток передачи событий был бы заблокирован, что сделало бы недоступной нижнюю кнопку с названием «Нажми меня». В нашем же случае пользовательский интерфейс остается отзывчивым даже во время выполнения фоновой задачи. Вы можете сами в этом убедиться.

С помощью объекта `SwingWorker` можно делать и более сложные вещи — например, последовательно обновлять индикатор выполнения. Этот вариант его использования рассматривается в документации (во вводной части)*.

Использование механизма передачи событий или объекта `SwingWorker` не всегда является оправданным. Например, в простых приложениях все действия обычно выполняются в ответ на события пользовательского интерфейса, что само по себе позволяет безопасно изменять графические компоненты. Но если вы создаете поток, в котором выполняются долгоиграющие задачи, такие как загрузка данных или взаимодействие по сети, вам следует учитывать вышеописанные риски.

---

* **docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html**

Глава 17

ИСПОЛЬЗОВАНИЕ КОМПОНЕНТОВ БИБЛИОТЕКИ SWING

В предыдущей главе мы обсудили целый ряд концепций, включая принцип построения пользовательского интерфейса и работу основных его частей. Вы уже должны иметь представление о том, что такое компоненты, контейнеры и диспетчеры компоновки, как они взаимодействуют друг с другом при выводе на экран, как они связываются с остальным кодом приложения с помощью событий.

Теперь, закончив с основами и как следует подготовившись, можно перейти к самому интересному — созданию настоящих приложений с помощью библиотеки Swing. Мы рассмотрим большинство компонентов из пакета `javax.swing`, научимся использовать их в апплетах и настольных программах, а также узнаем, как создать свой собственный компонент. Мы подготовили для вас много кода и интересных примеров.

Эта тема слишком обширная, чтобы пытаться уместить ее в одну главу. Поэтому здесь мы сосредоточимся на основных элементах пользовательского интерфейса, а уже в следующей главе перейдем к рассмотрению более сложных вопросов, таких как работа с текстом, древовидные списки, таблицы и создание собственных компонентов.

Кнопки и метки

Начнем с самых простых компонентов — кнопок и меток. Если честно, это не самая богатая тема для разговора. Если вы видели одну кнопку, можете считать, что вы видели и все остальные (вспомните кнопки в приложениях HelloJava3 и HelloJava4 из главы 2). При нажатии кнопка генерирует событие `ActionEvent`, которое передается в качестве аргумента для метода `actionPerformed()`, принадлежащего зарегистрированной реализации интерфейса `ActionListener`.

У кнопок есть еще одно свойство, о котором стоит упомянуть, хотя это касается любых компонентов, генерирующих события типа `ActionEvent`

(например, элементов меню). Платформа Java позволяет присваивать кнопкам специальные «команды действия» (action commands). На самом деле в них нет ничего интересного — это обычные строки, которые идентифицируют компонент, приславший событие. По умолчанию команда кнопки совпадает с текстом, который та отображает; вы можете извлечь ее из объекта `ActionEvent`, чтобы определить, какая именно кнопка была нажата. Хотя обычно эта информация доступна в контексте вашего слушателя.

Получить команду действия можно с помощью метода `getActionCommand()`. Следующий код проверяет, помечена ли нажатая кнопка текстом «Да»:

```
public void actionPerformed(ActionEvent e){
   if (e.getActionCommand().equals("Да") {
      //пользователь нажал кнопку "Да"
      //делаем что-нибудь
      ...
   }
}
```

«Да» — это строка, а не команда как таковая. Чтобы изменить команду действия, можно воспользоваться методом `setActionCommand()` из объекта `ActionEvent`. Следующий код устанавливает для кнопки `myButton` команду «подтвердить»:

```
myButton.setActionCommand("подтвердить");
```

Вам стоит приучить себя к тому, что команды нужно устанавливать вручную; это поможет избежать проблем в ситуациях, когда названия кнопок изменяются без вашего ведома (например, в результате локализации). Если полагаться только на отображаемый текст, ваш код перестанет работать, как только кто-то внесет изменения в пользовательский интерфейс. Имейте в виду, что у пользователя из Франции кнопка может называться не «Да», а «Oui».

Библиотека Swing позволяет выводить на кнопках не только текст, но и изображения. Один из конструкторов класса `JButton` принимает объект `Icon`, который умеет выводить на экран графические файлы. Кнопки позволяют выводить текст и изображение вместе или по отдельности. Для загрузки изображений, которые потом можно передавать объекту `Icon`, существует удобный класс `ImageIcon`. В следующем примере показано, как это все работает:

```
//файл: PictureButton.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class PictureButton {
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      Icon icon = new ImageIcon("rhino.gif");
      JButton button = new JButton(icon);
      button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            System.out.println("Кнопка нажата");
         }
      });
      frame.getContentPane().add( button );
      frame.pack();
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setVisible(true);
   }
}
```

В этом примере на основе файла *rhino.gif* создается объект `ImageIcon`, который, в свою очередь, используется для создания кнопки `JButton`. Все это выводится внутри контейнера `JFrame`. Здесь также демонстрируется принцип использования реализации `ActionListener` в виде анонимного внутреннего класса. Если кнопки выглядят довольно скучными, то что уж говорить о метках. Объект `JLabel` — это компонент, который выводит текстовую строку или изображение. Он не поддерживает какие-то специальные события; все, что вы можете с ним сделать — это указать, по какому краю области отображения метки будет выравниваться текст. Как и в случае с кнопками, метки позволяют использовать объект `Icon` для вывода графических файлов. В следующем коде создаются разные варианты объекта `JLabel`:

```
// стандартное выравнивание (по центру)
JLabel label1 = new JLabel("львы");
// выравнивание по левому крают
JLabel label2 = new JLabel("тигры", SwingConstants.LEFT);
// метка без такста, стандартное выравнивание
JLabel label3 = new JLabel();
```

```
// загружаем изображение
Icon icon = new ImageIcon("rhino.gif");
// создаем графическую метку
JLabel label4 = new JLabel(icon);
// присваиваем текст метке label3
label3.setText("и медведи");
// устанавливаем выравнивание
label3.setHorizontalAlignment(SwingConstants.RIGHT);
```

Константы, определяющие выравнивание, объявлены в интерфейсе `SwingConstants`.

Мы создали несколько меток, используя разные конструкторы и методы. Чтобы вывести метку на экран, просто добавьте ее в контейнер с помощью метода `add()`. Вы можете изменять и другие свойства метки, такие как шрифт или цвет, используя методы `setFont()` и `setBackground()` из класса `Component`, далекого предка `JLabel`.

Если метки настолько простые, зачем же они нам нужны? Почему бы не выводить текст непосредственно на контейнере? Не стоит забывать, что объект `JLabel` является потомком класса `JComponent` и, следовательно, имеет стандартный набор методов для изменения шрифта и цвета. Кроме того, управление метками можно полностью доверить диспетчеру компоновки, что делает их более гибкими, чем обычный текст, который выводится в произвольном месте контейнера процедурным способом. К слову о компоновке: если изменить текст метки с помощью метода `setText()`, изменится и ее предпочтительный размер. Но вам не нужно об этом беспокоиться, так как в таком случае контейнер автоматически пересмотрит расположение своих компонентов.

Разметка текста в кнопках и метках с помощью формата HTML

Среди прочих интересных особенностей библиотеки Swing можно выделить возможность форматирования текста в метках и кнопках с помощью HTML. В следующем примере показано, как создать кнопку с разметкой, словно у веб-страницы:

```
JButton button = new JButton(
   "<html>"
   + "П<font size=-1>ИШЕМ<font size=+0> "
   + "К<font size=-1>АПИТЕЛЬЮ");
```

Старые версии платформы Java не очень хорошо поддерживали сложное форматирование. Но с выходом JDK 1.4 стали доступны основные возможности языка HTML, включая такие продвинутые вещи как изображения и таблицы. На рис. 17.1 показано, как можно вывести текст в виде таблицы.

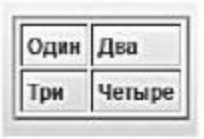

Рис. 17.1. Вывод HTML-таблицы внутри кнопки

В примере, показанном на рис. 17.2, с помощью HTML-тега выводится изображение.



Рис. 17.2. Вывод изображения внутри кнопки с помощью HTML-тега

Код для вывода этих двух элементов выглядит следующим образом:

```
String html=
   "<html><table border=1>"
   +"<tr><td>Один</td><td>Два</td></tr>"
   +"<tr><td>Три</td><td>Четыре</td></tr>"
   +"</table>";
JButton button = new JButton(html);
String html2 =
   "<html><h3>Большая панда</h3>"
   +"<img src=\"http://is.gd/AMpif5\">";
Jbutton button2 = new JButton(html2);
```

Флажки и переключатели

Флажок (`JCheckBox`) — это, по сути, тумблер с меткой. При каждом щелчке мышью он меняет свое состояние с «установленного» на «сброшенное» и обратно. В библиотеке Swing флажок реализован в виде спе-

циальной кнопки. Переключатели (`JRadioButton`) похожи на флажки, но обычно их используют группами. Установить переключатель можно только в одно положение (то есть, выбирается только один элемент `JRadioButton` из всей группы, тогда как все остальные автоматически сбрасываются). Название «Radio button» (радио-кнопка) было выбрано не случайно: когда-то похожие кнопки использовались в старых автомобилях для управления радиоприемником.

Флажки и переключатели представлены экземплярами классов `JCheckBox` и `JRadioButton` соответственно. Переключатели можно группировать вместе с помощью еще одного объекта, `ButtonGroup`. К этому моменту вы уже, вероятно, понимаете, что к чему, и можете легко управиться с этими классами без нашей помощи. В следующем примере мы проиллюстрируем разные подходы к управлению состоянием компонентов, а также раскроем некоторые дополнительные возможностей контейнеров.

При нажатии объект `JCheckBox` генерирует сразу два события: первое, `ItemEvent`, как флажок, и второе, `ActionEvent`, как разновидность кнопки. У этого элемента управления есть одна особенность: обычно его состояние становится важным только в момент, когда пользователь выполняет какое-то действие. Например, то, какие флажки устанавливаются во время заполнения формы, имеет значение только после нажатия кнопки **Отправить**.

Следующее приложение под названием DriveThrough позволяет подобрать себе обед в меню закусочной, как показано на рис. 17.3.

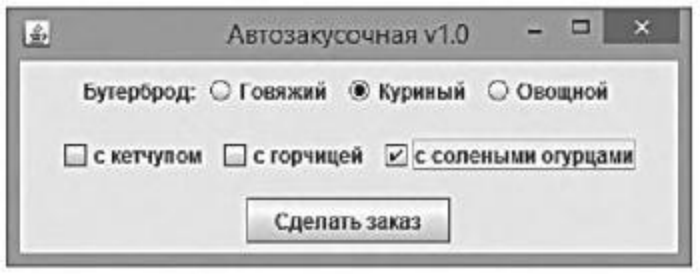

Рис. 17.3. Приложение DriveThrough

Приложение DriveThrough выводит результаты при нажатии кнопки **Сделать заказ**. Таким образом, мы можем игнорировать все события, создаваемые переключателями и флажками, и отслеживать только те объекты `ActionEvent`, которые генерируются нашей кнопкой:

```
//файл: DriveThrough.java
import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;
public class DriveThrough
{
   public static void main(String[] args) {
      JFrame frame = new JFrame("Автозакусочная v1.0");
      JPanel entreePanel = new JPanel();
      JLabel sandwich = new JLabel("Бутерброд:");
      entreePanel.add(sandwich);
      final ButtonGroup entreeGroup = new ButtonGroup();
      JRadioButton radioButton;
      entreePanel.add(radioButton = new JRadioButton("Говяжий"));
      radioButton.setActionCommand("Говяжий");
      entreeGroup.add(radioButton);
      entreePanel.add(radioButton = new JRadioButton("Куриный"));
      radioButton.setActionCommand("Куриный");
      entreeGroup.add(radioButton);
      entreePanel.add(radioButton = new JRadioButton("Овощной", true));
      radioButton.setActionCommand("Овощной");
      entreeGroup.add(radioButton);
      final JPanel condimentsPanel = new JPanel();
      condimentsPanel.add(new JCheckBox("с кетчупом"));
      condimentsPanel.add(new JCheckBox("с горчицей"));
      condimentsPanel.add(new JCheckBox("с солеными огурцами"));
      JPanel orderPanel = new JPanel();
      JButton orderButton = new JButton("Сделать заказ");
      orderPanel.add(orderButton);
      // с версии 5.0 является опциональным
      Container content = frame.getContentPane();
      content.setLayout(new GridLayout(3, 1));
      content.add(entreePanel);
      content.add(condimentsPanel);
      content.add(orderPanel);
      orderButton.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            String entree =
               entreeGroup.getSelection().getActionCommand();
            System.out.println(entree + " бутерброд");
            Component[] components = condimentsPanel.getComponents();
            for ( Component c : components ) {
               JCheckBox cb = (JCheckBox)c;
```

```
                if (cb.isSelected())
                    System.out.println(cb.getText());
            }
        }
    });
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setSize(400, 150);
    frame.setVisible(true);
  }
}
```

Приложение DriveThrough состоит из трех панелей. Переключатели, сгруппированные с помощью объекта `ButtonGroup`, находятся на панели `entreePanel`. Мы поместили их в одну группу с помощью метода `add()`, чтобы сделать взаимоисключающими. `ButtonGroup` — не совсем обычный объект. С виду он похож на контейнер или компонент, но на самом деле это всего лишь вспомогательный класс, который позволяет выбирать одновременно только один переключатель.

В этом примере вам нужно выбрать для своего бутерброда что-то одно: говядину, курицу или овощи. Флажки, отвечающие за приправу, не находятся в одной группе, поэтому вы можете устроить себе любое сочетание из кетчупа, горчицы и соленых огурцов.

При нажатии кнопки **Сделать заказ** в метод `actionPerformed()` нашей реализации `ActionListener` поступает событие `ActionEvent`. В этот момент мы собираем информацию о переключателях и флажках и выводим ее в командную строку. Метод `actionPerformed()` просто считывает состояние разных компонентов. Ссылки на элементы управления можно сохранять по-разному; в этом примере мы демонстрируем два способа. Сначала мы выясняем, какие элементы были выбраны. Для этого мы используем метод `getSelection()` из объекта `ButtonGroup`. Из полученного результата, `ButtonModel`, сразу же вызывается метод `getActionCommand()`. В итоге к нам попадает одна из тех команд действия, которые мы устанавливали при создании переключателей. В ней содержится название бутерброда — именно то, что нам нужно.

Чтобы узнать, какие приправы были выбраны, используется более сложная процедура. Проблема в том, что здесь варианты не являются взаимоисключающими и у нас нет удобной группы `ButtonGroup`. Сначала мы получаем список компонентов, которые хранятся внутри объекта `JPanel`. Метод `getComponents()` возвращает массив ссылок на дочерние

объекты контейнера. Чтобы вывести результаты, мы пройдемся по нему с помощью цикла. Прежде чем определять состояние компонента с помощью метода `isSelected()`, мы приводим его к типу `JCheckBox`. Если бы у нас в массиве хранились разные виды элементов управления, их настоящий тип можно было бы узнать с помощью оператора `instanceof`. Мы могли бы как-то систематизировать наши компоненты — например, с помощью ассоциативного массива, ключами для которого выступают имена объектов.

Обычные и раскрывающиеся списки

Компоненты `JList` и `JComboBox` — это еще один этап в развитии графических элементов управления. Список предоставляет набор вариантов, среди которых пользователь может выбрать один или сразу несколько. Обычно не все пункты выводятся одновременно; полосы прокрутки позволяют переходить к элементам, которые изначально не видны на экране. Чтобы выбрать пункт, по нему нужно щелкнуть мышью. Если удерживать клавишу **Shift**, можно выбрать несколько последовательных элементов списка. Для множественного выбора отдельных пунктов предусмотрена клавиша **Ctrl** (или **Command**, в операционной системе OS X).

Раскрывающийся список — это нечто среднее между обычным списком и полем ввода. Изначально он выводит только одну строчку текста (иногда с изображением), справа от которой находится стрелка, указывающая вниз. Если щелкнуть мышью по этой стрелке, список раскроется и отобразит набор вариантов, один из которых можно выбрать. После того как пользователь сделает свой выбор, список обратно закроется, а в поле ввода будет отображаться выбранный вариант.

Как и остальные компоненты библиотеке Swing, эти два вида списков состоят из модели данных и визуальной части. Кроме того, объект `JList` поддерживает механизм, который управляет способом выбора элементов.

Сходство компонентов `JList` и `JComboBox` обусловлено тем, что они используют похожие модели данных. В сущности, оба они представляют собой массивы элементов с возможностью выбора. Это видно и по их внутреннему устройству: тип модели данных объекта `JComboBox` является потомком класса, который предоставляет модель данных для компонента `JList`. Это родство проиллюстрировано в следующем примере.

Приложение, представленное ниже, создает окно с двумя списками (обычным и раскрывающимся) и кнопкой. Оба списка используют одну и ту же модель данных. При нажатии кнопки приложение выводит список выбранных пунктов (см. рис. 17.4).

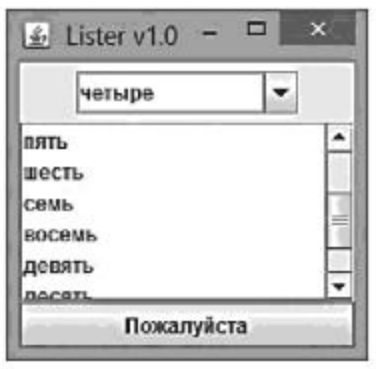

Рис. 17.4. Компоненты `JList` и `JComboBox` с общей моделью данных

Ниже представлен код этого приложения:

```
//файл: Lister.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Lister {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Lister v1.0");
      // создаем раскрывающийся список
      String [] items = { "один", "два", "три",
                  "четыре", "пять", "шесть",
                  "семь", "восемь", "девять",
                  "десять", "одиннадцать",
                  "двенадцать" };
      JComboBox comboBox = new JComboBox(items);
      comboBox.setEditable(true);
      // создаем список с той же моделью данных
      final JList list = new JList(comboBox.getModel());
      // создаем кнопку; при ее нажатии выводим
      // выбранные элементы списка
      JButton button = new JButton("Пожалуйста");
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            Object[] selection = list.getSelectedValues();
            System.out.println("-----");
            for ( Object o : selection )
               System.out.println( o );
```

```
    }
});
// размещаем элементы на главной панели
// с версии 5.0 является опциональным
Container c = frame.getContentPane();
JPanel comboPanel = new JPanel();
comboPanel.add(comboBox);
c.add(comboPanel, BorderLayout.NORTH);
c.add(new JScrollPane(list), BorderLayout.CENTER);
c.add(button, BorderLayout.SOUTH);
frame.setSize(200, 200);
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setVisible(true);
  }
}
```

Раскрывающийся список создается на основе массива строк. На самом деле все немного сложнее: из предоставленных вами элементов типа `String` конструктор объекта `JComboBox` создает модель данных и делает так, чтобы она использовалась в дальнейшем. Затем с помощью этой же модели создается компонент `JList`. Это вполне корректный подход, так как `ComboBoxModel` (модель данных раскрывающегося списка) является наследником класса `ListModel` (модели данных списка `JList`).

Обработчик событий `ActionEvent`, генерируемых кнопкой, просто выводит выбранные пункты списка, предварительно извлекая их с помощью метода `getSelectedValues()`. На самом деле массив, который возвращает этот метод, содержит объекты типа `Object`, а не строки. Элементы списков, как и многие другие компоненты в библиотеке Swing, не ограничиваются поддержкой одного лишь текста. Вы можете выводить в них изображения, программно созданные рисунки или комбинацию текста и графических элементов. Но в этом примере мы просто выводим строки.

Вы, наверное, ожидали, что выбор пункта в раскрывающемся списке будет приводить к выделению аналогичного пункта в компоненте `JList`. В библиотеке Swing за это отвечает отдельная модель на основе интерфейса `ListSelectionModel`. Механизм выбора элементов может отличаться: объект `JComboBox` позволяет выбрать только один пункт, тогда как список `JList` допускает множественный выбор. Таким образом, несмотря на общую модель данных, эти два списка имеют разные механизмы выделения элементов.

Мы сделали так, чтобы раскрывающийся список можно было редактировать. По умолчанию он доступен только для чтения: пользователь может выбирать один из элементов, но не более того. В нашем случае добавление текста происходит так же, как в обычном поле ввода. Редактируемый раскрывающийся список удобен в ситуациях, когда помимо некоторых универсальных вариантов должна присутствовать возможность ввода произвольного текста. Если вы хотите, чтобы пользователь мог выбирать только из заранее подготовленных пунктов, редактирование лучше не разрешать.

В нашем примере припрятан один замечательный класс, о котором стоит упомянуть отдельно. Это `JScrollPane`. Мы воспользовались им, когда добавляли компонент `JList` в главное окно. Это простая обертка вокруг класса `Component`, которая в случае необходимость предоставляет полосы прокрутки. «Необходимость» возникает в момент, когда предпочтительный размер компонента (возвращаемый методом `getPreferredSize()`) начинает превышать текущий размер объекта `JScrollPane`. В предыдущем примере полосы прокрутки появлялись, когда для вывода списка уже не хватало свободного места.

Вы можете помещать внутрь объекта `JScrollPane` любые компоненты, включая поверхности для рисования, изображения или панели со сложным пользовательским интерфейсом. Более подробно об этом классе мы поговорим позже в этой главе, а начиная с главы 18 мы начнем активно его использовать в сочетании с компонентами для работы с текстом.

Поле ввода со счетчиком

Компоненты `JList` и `JComboBox` дают возможность пользователю выбирать из нескольких вариантов. Объект `JComboBox` обладает определенной гибкостью, ведь его можно сделать редактируемым, но в целом при использовании этих двух списков вы ограничены заранее фиксированным набором пунктов. С выходом платформы Java 1.4 в библиотеке Swing появился новый компонент под названием `JSpinner` (поле ввода со счетчиком), который подходит для представления больших или бесконечных последовательностей значений, таких как числа или даты. Это близкий родственник раскрывающегося списка; но вместо того чтобы раскрываться и предоставлять готовые варианты, в правой части поля ввода он показывает две маленькие стрелки («вверх» и «вниз»), с помощью которых можно перебирать диапазон значений (см. рис. 17.5).

Как и раскрывающийся список, поле ввода со счетчиком можно сделать редактируемым, чтобы пользователь мог вводить данные напрямую.

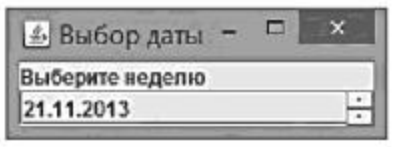

Рис. 17.5. Внешний вид программы для выбора даты

Библиотека Swing содержит три разновидности компонента `JSpinner`, каждая из которых представлена отдельной моделью данных: `SpinnerListModel`, `SpinnerNumberModel` и `SpinnerDateModel`.

Модель `SpinnerListModel` работает по принципу раскрывающегося списка, предлагая фиксированный набор объектов:

```
String [] options = new String [] { "маленький", "средний", "большой",
"огромный" };
SpinnerListModel model = new SpinnerListModel( options );
JSpinner spinner = new JSpinner( model );
```

В любой момент из модели можно извлечь ее текущее значение:

```
String value = (String)model.getValue();
```

Чтобы получать значение всякий раз, когда пользователь его изменяет, можно зарегистрировать объект-слушатель `ChangeListener`. Модель `SpinnerListModel` может автоматически сверять введенное пользователем значение с набором допустимых вариантов. Этим она отличается от других моделей, которые принимают любые данные, подходящие по типу (например, числа или даты). Стоит сказать, что все это возможно только в том случае, если компонент `JSpinner` является редактируемым.

Модель `SpinnerNumberModel` предназначена для вывода числовых типов. Для нее можно указать начально, минимальное и максимальное значения:

```
double initial=5.0, min=0.0, max=10.0, increment=0.1;
SpinnerNumberModel model =
   new SpinnerNumberModel( initial, min, max, increment );
JSpinner spinner = new JSpinner(model);
```

Здесь мы создали поле ввода со счетчиком, которое начинается с числа `5,0`, и позволяет выбирать значения в диапазоне от `0` до `10` с шагом `0,1`. Метод `getNumber()` из объекта `SpinnerNumberModel` возвращает текущий выбор.

Наверное, самой интересной моделью данных компонента `JSpinner` является `SpinnerDateModel`, с помощью которой пользователи могут указывать даты в календаре, перебирая определенные промежутки времени. Как и `SpinnerNumberModel`, она принимает диапазон значений, только здесь они имеют тип `Date`, а в качестве шага используются константы из класса `java.util.Calendar`: `Calendar.DAY`, `Calendar.WEEK` и т. д. Следующее приложение под названием `DateSelector` создает поле ввода со счетчиком, которое показывает текущую дату. Оно позволяет перемещаться на шесть месяцев вперед и назад (то есть, диапазон равен одному году) с промежутком в неделю. Для модели зарегистрирован слушатель `ChangeListener`, который выводит данные по мере их изменения:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.util.*;
public class DateSelector {
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Выбор даты");
      Calendar now = Calendar.getInstance( );
      Calendar earliest = (Calendar)now.clone();
      earliest.add( Calendar.MONTH, -6 );
      Calendar latest = (Calendar)now.clone();
      latest.add( Calendar.MONTH, 6 );
      SpinnerModel model = new SpinnerDateModel(
         now.getTime(), earliest.getTime(), latest.getTime(),
         Calendar.WEEK_OF_YEAR);
      final JSpinner spinner = new JSpinner(model);
      // Отключаем встроенный редактор даты
      spinner.setEditor(new JSpinner.DefaultEditor(spinner) );

      model.addChangeListener( new ChangeListener() {
         public void stateChanged(ChangeEvent e) {
         System.out.println( ((SpinnerDateModel)e.getSource()).getDate() );
         }
      } );
      frame.getContentPane().add( "North", new JLabel("Выберите неделю") );
      frame.getContentPane().add( "Center", spinner );
      frame.pack();
```

```
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible(true);
    }
}
```

Как мы уже говорили, модель `SpinnerCalendarModel` похожа на `SpinnerNumberModel`, только вместо чисел она работает с объектами типа `Date` и в качестве интервала использует константы из класса `Calendar`. Чтобы получить дату, мы создали объект `Calendar` и воспользовались его методом `getTime()`. В этом примере с помощью метода `add()` мы устанавливаем минимальное и максимальное значения (шесть месяцев в каждом направлении). В табл. 17.1 представлены поля класса `Calendar`, которые можно использовать в качестве интервала.

Табл. 17.1. Константы из класса `Calendar`

Поле	Интервал
`Calendar.MILLISECOND`	Одна миллисекунда
`Calendar.SECOND`	Одна секунда
`Calendar.MINUTE`	Одна минута
`Calendar.HOUR Calendar.HOUR_OF_DAY`	Один час
`Calendar.AM_PM`	Утро/вечер
`Calendar.DAY_OF_WEEK Calendar.DAY_OF_MONTH Calendar.DAY_OF_YEAR`	Один день
`Calendar.MONTH`	Один месяц
`Calendar.YEAR`	Один год
`Calendar.ERA`	До н.э. и н.э. (в григорианском календаре)

Чтобы увеличить или уменьшить значение, модель `SpinnerDateModel` передает в метод `add()` числа 1/–1 в сочетании с соответствующими константами. Как видно из табл. 17.1, некоторые константы имеют одинаковый интервал.

Обрамление

Любой компонент библиотеки Swing может иметь декоративное обрамление. Класс `JComponent` содержит метод `setBorder()`, которому можно передать подходящую реализацию интерфейса `Border`.

В пакете `javax.swing.border` содержится множество готовых обрамлений. Вы можете указать экземпляр одного из этих классов с помощью

метода `setBorder()`, принадлежащего интересующему вас компоненту, но можно сделать еще проще.

Класс `BorderFactory` позволяет создавать любые обрамления с помощью статических «фабричных» методов. Вот как это выглядит:

```
JLabel labelTwo = new JLabel("У меня гравированное обрамление.");
labelTwo.setBorder(BorderFactory.createEtchedBorder());
```

Метод `setBorder()` есть у всех элементов управления — от обычных меток и кнопок до таблиц и сложных компонентов для работы с текстом (о которых мы поговорим в главе 18).

Несмотря на свое удобство, класс `BorderFactory` не позволяет полностью сконфигурировать любой вид обрамления. Например, если вы хотите сделать гравированное обрамление выпуклым, вам придется использовать конструктор класса `EtchedBorder`:

```
JLabel labelTwo = new JLabel("У меня выпуклое гравированное обрамление.");
labelTwo.setBorder( new EtchedBorder(EtchedBorder.RAISED) );
```

Ниже кратко описываются реализации интерфейса `Border`:

`BevelBorder`

Это обрамление может иметь выпуклые или вдавленные края; создает иллюзию объема.

`SoftBevelBorder`

То же самое, что `BevelBorder`, только тоньше.

`EmptyBorder`

Ничего не рисует, но занимает место на экране. Можете использовать его, чтобы элементы интерфейса не находились слишком тесно друг к другу.

`EtchedBorder`

Вдавленное гравированное обрамление выглядит как прямоугольник, вырубленный на каменной плите. Выпуклый вариант похож на панель, которая выступает на поверхности экрана.

`LineBorder`

Рисует вокруг компонента обычный прямоугольник. В конструкторе объекта `LineBorder` можно выбрать цвет и толщину линий.

`MatteBorder`

Более гибкая версия `LineBorder`. Вы можете изменять цвет и размер обрамления по левому, правому, верхнему и нижнему краям. Объект `MatteBorder` можно также использовать в качестве рамки для реализаций интерфейса `Icon` — например для изображений (`ImageIcon`).

`TitledBorder`

Обычное обрамление с заголовком. На самом деле рамку рисует другой объект, а `TitledBorder` просто выводит текст. Вы можете указать местоположение и шрифт заголовка, а также край, по которому он будет выравниваться. Этот вид обрамления особенно полезен для группирования разных наборов компонентов в сложных пользовательских интерфейсах.

`CompoundBorder`

Состоит сразу из двух отдельных обрамлений. Хорошо подходит в тех случаях, когда компонент нужно поместить внутрь объекта `EmptyBorder`, но при этом хочется нарисовать вокруг него какую-нибудь декоративную рамку (например, `EtchedBorder` или `MatteBorder`).

В следующем примере показаны разные виды обрамлений. Но это только небольшая выборка; на самом деле вам доступно значительно больше вариантов. К тому же здесь выводятся обычные текстовые метки, хотя обрамление можно установить для любого компонента библиотеки Swing. Итоговый результат показан на рис. 17.6.

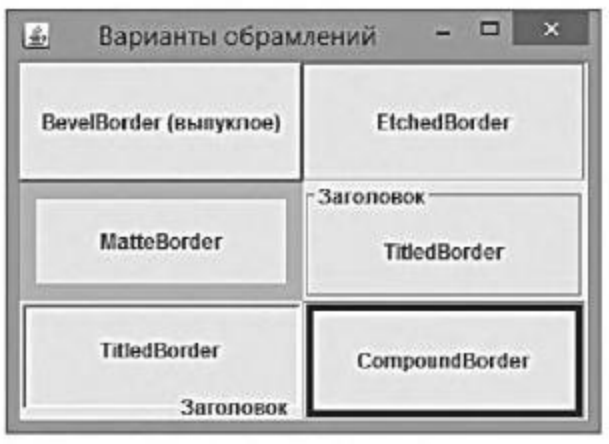

Рис. 17.6. Набор разных обрамлений

А вот и исходный код:

```
//файл: Borders.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class Borders {
```

```
public static void main(String[] args) {
   // создаем контейнер JFrame, который будет хранить
   // все остальные элементы
   JFrame frame = new JFrame("Обрамления");
   // создаем метки с обрамлениями
   int center = SwingConstants.CENTER;
   JLabel labelOne = new JLabel("BevelBorder (выпуклое)", center);
   labelOne.setBorder(
BorderFactory.createBevelBorder(BevelBorder.RAISED));
   JLabel labelTwo = new JLabel("EtchedBorder", center);
labelTwo.setBorder(BorderFactory.createEtchedBorder());
   JLabel labelThree = new JLabel("MatteBorder", center);
   labelThree.setBorder(
      BorderFactory.createMatteBorder(10, 10, 10, 10, Color.pink));
   JLabel labelFour = new JLabel("TitledBorder", center);
   Border etch = BorderFactory.createEtchedBorder();
   labelFour.setBorder(
      BorderFactory.createTitledBorder(etch, "Заголовок"));
   JLabel labelFive = new JLabel("TitledBorder", center);
   Border low = BorderFactory.createLoweredBevelBorder();
   labelFive.setBorder(
      BorderFactory.createTitledBorder(low, "Заголовок",
      TitledBorder.RIGHT, TitledBorder.BOTTOM));
   JLabel labelSix = new JLabel("CompoundBorder", center);
   Border one = BorderFactory.createEtchedBorder();
   Border two =
      BorderFactory.createMatteBorder(4, 4, 4, 4, Color.blue);
labelSix.setBorder(BorderFactory.createCompoundBorder(one, two));
   // размещаем элементы на главной панели
   // с версии 5.0 является опциональным
      Container c = frame.getContentPane();
      c.setLayout(new GridLayout(3, 2));
      c.add(labelOne);
      c.add(labelTwo);
      c.add(labelThree);
      c.add(labelFour);
      c.add(labelFive);
      c.add(labelSix);
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.pack();
      frame.setVisible(true);
   }
}
```

Меню

Компонент `JMenu` представляет собой стандартное выпадающее меню с фиксированным названием. Он поддерживает вложенность, благодаря чему вы можете реализовывать сложные иерархии. В библиотеке Swing меню является полноценным элементом управления. Вы можете поместить его в любое место окна, где допускается нахождение графического компонента. Есть еще один класс, `JMenuBar`, который позволяет размещать меню в рамках привычной горизонтальной строки. Это тоже полноценный компонент, поэтому его можно добавить в любой контейнер: сверху, снизу или по центру. Но имейте в виду, что посреди окна вместо меню более логично было бы выводить раскрывающийся список.

Для пунктов меню можно устанавливать изображения и сочетания клавиш; они могут принимать вид флажков или переключателей. В сущности это разновидность кнопок. При выборе пунктов меню генерируются события, перехватывать которые можно с помощью зарегистрированных реализаций интерфейса `ActionListener`.

Управлять меню с помощью клавиатуры можно двумя способами. Первый подразумевает использование мнемонических символов (обычно они подчеркиваются в названии): если вы наберете такой символ, удерживая клавишу **Alt**, меню раскроется так, как будто вы щелкнули по нему мышью. Мнемонические символы могут быть и у отдельных пунктов — после раскрытия меню вы можете выбирать их таким же образом.

Даже если пункт меню не отображается на экране, его можно выбрать с помощью сочетаний клавиш. Ярким примером является сочетание **Ctrl+C**, которое часто привязано к команде **Копировать** из меню **Правка**.

В следующем примере продемонстрировано использование трех разных компонентов `JMenu` в рамках одной строки меню. Первый компонент под названием **Утварь** содержит несколько пунктов, вложенное меню, разделитель и команду **Выйти**, у которой есть как мнемонический символ, так и комбинация клавиш. Во втором меню, **Специи**, находится несколько пунктов, которые выглядят и ведут себя как флажки. Меню **Сыр** содержит переключатели.

Данное приложение показано на рис. 17.7, там же вы можете увидеть один из элементов меню в раскрытом состоянии. Закрыть окно можно

с помощью команды **Выйти** из меню **Утварь** (или нажав сочетание клавиш **Ctrl+Q**).

```
//файл: DinnerMenu.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DinnerMenu
{
   public static void main(String[] args) {
      JFrame frame = new JFrame("Обеднее меню");
      // создаем меню Утварь
      JMenu utensils = new JMenu("Утварь");
      utensils.setMnemonic('У');
      utensils.add(new JMenuItem("Вилка"));
      utensils.add(new JMenuItem("Нож"));
      utensils.add(new JMenuItem("Ложка"));
      JMenu hybrid = new JMenu("Гибридные");
      hybrid.add(new JMenuItem("Вложка"));
      hybrid.add(new JMenuItem("Ножилка"));
      hybrid.add(new JMenuItem("Крошилка"));
      utensils.add(hybrid);
      utensils.addSeparator();
      // делаем кое-что интересное с пунктом Выйти
      JMenuItem quitItem = new JMenuItem("Выйти");
      quitItem.setMnemonic('В');
      quitItem.setAccelerator(
         KeyStroke.getKeyStroke(KeyEvent.VK_Q, Event.CTRL_MASK));
      quitItem.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) { System.exit(0); }
      });
      utensils.add(quitItem);
      // создаем меню Специи
      JMenu spices = new JMenu("Специи");
      spices.setMnemonic('С');
      spices.add(new JCheckBoxMenuItem("Тимьян"));
      spices.add(new JCheckBoxMenuItem("Розмарин"));
      spices.add(new JCheckBoxMenuItem("Душица", true));
      spices.add(new JCheckBoxMenuItem("Фенхель"));
      // создаем меню Сыр
      JMenu cheese = new JMenu("Сыр");
```

```
        cheese.setMnemonic('ы');
        ButtonGroup group = new ButtonGroup();
        JRadioButtonMenuItem rbmi;
        rbmi = new JRadioButtonMenuItem("Обычный", true);
        group.add(rbmi);
        cheese.add(rbmi);
        rbmi = new JRadioButtonMenuItem("Чеддер");
        group.add(rbmi);
        cheese.add(rbmi);
        rbmi = new JRadioButtonMenuItem("Голубой");
        group.add(rbmi);
        cheese.add(rbmi);
        // создаем строку меню и добавляем ее в JFrame
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(utensils);
        menuBar.add(spices);
        menuBar.add(cheese);
        frame.setJMenuBar(menuBar);
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize(200,200);
        frame.setVisible(true);  }
}
```

Да, мы понимаем, что пункт **Выйти** плохо сочетается меню **Утварь**. Если вас это так беспокоит, можете в конце этого раздела вернуться назад и добавить меню **Файл**.

В построении меню нет ничего сложного: достаточно просто создать объект `JMenu` и указать для него название. По аналогии с компонентами `JButton` и `JLabel`, текст меню может содержать простую разметку в формате HTML. Подготовив объект `JMenu`, вы можете добавлять в него отдельные пункты (`JMenuItem`) или другие меню, которые будут выводиться как вложенные. Рассмотрим этот процесс на примере создания меню **Утварь**:

```
JMenu utensils = new JMenu("Утварь");
utensils.setMnemonic('У');
utensils.add(new JMenuItem("Вилка"));
utensils.add(new JMenuItem("Нож"));
utensils.add(new JMenuItem("Ложка"));
JMenu hybrid = new JMenu("Гибридные");
hybrid.add(new JMenuItem("Вложка"));
```

```
hybrid.add(new JMenuItem("Ножилка"));
hybrid.add(new JMenuItem("Крошилка"));
utensils.add(hybrid);
```

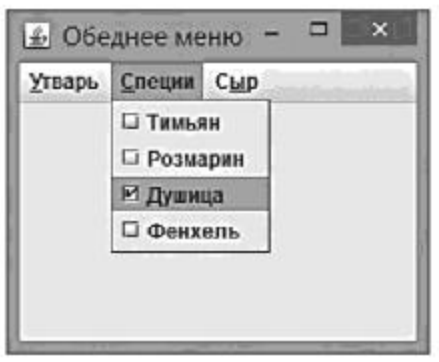

Рис. 17.7. Приложение DinnerMenu

Во второй строчке мы устанавливаем для этого меню мнемонический символ. Если бы пользовательский интерфейс был на английском языке, мы бы могли указать одну из статических констант из класса `KeyEvent`, в котором содержатся идентификаторы всех латинских клавиш на клавиатуре.

С помощью всего лишь одной строчки в меню можно добавить симпатичный разделитель:

```
utensils.addSeparator();
```

На пункте меню **Выйти** следует остановиться отдельно. Сначала мы создает компонент `JMenuItem` и назначаем ему мнемонический символ, как мы это делали с меню **Утварь**:

```
JMenuItem quitItem = new JMenuItem("Выйти");
quitItem.setMnemonic('B');
```

Затем мы создаем для него комбинацию клавиш, используя класс под названием `KeyStroke`:

```
quitItem.setAccelerator(
   KeyStroke.getKeyStroke(KeyEvent.VK_Q, Event.CTRL_MASK));
```

И в конце, чтобы выбор этого пункта приводил к каким-то действиям, мы регистрируем анонимный объект-слушатель:

```
quitItem.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { System.exit(0); }
});
```

При выборе пункта меню **Выйти** наш слушатель закрывает приложение.

С меню **Специи** мы сделали все то же самое, только вместо обычных пунктов типа `JMenuItem` добавили элементы `JCheckBoxMenuItem`. В итоге пункты этого меню ведут себя как флажки.

Со следующим меню, **Сыр**, все немного сложнее. Чтобы пункты можно было использовать в качестве переключателей, перед добавлением их нужно поместить в объект `ButtonGroup`.

В завершение мы помещаем наши экземпляры меню в объект `JMenuBar`, который выводит их на горизонтальной панели. Поскольку `JMenuBar` является самым обычным компонентом, мы могли бы вручную добавить его на главную панель контейнера `JFrame`. Но вместо этого мы воспользовались удобным методом под названием `setJMenuBar()`, который автоматически размещает меню в верхней части окна. Благодаря этому главная панель сама адаптируется к отображению строки меню, и нам не нужно вручную изменять ее компоновку и размеры.

Контекстные меню

В библиотеке Swing есть отличный компонент под названием `JPopupMenu`. Это контекстное меню, которое появляется в месте расположения указателя при нажатии соответствующей клавиши мыши (или клавиатуры). Обычно для этого используется правая клавиша, но в старых моделях компьютеров Mac с однокнопочной мышью для вызова контекстного меню нужно сделать щелчок с нажатой клавишей **Command**. На разных платформах это работает по-разному, но библиотека Swing сама выполняет все нужные проверки, так что это не должно отразиться на вашем коде.

В целом, если не считать отсутствия аргументов у конструктора `JPopupMenu()`, компонент `JPopupMenu` создается и наполняется так же, как любое другое меню. Главное отличие заключается в том, что его не нужно добавлять на панель `JMenuBar`. Контекстное меню появляется там, где это необходимо. До выхода платформы Java 5.0 этот процесс был немного громоздким; разработчику приходилось подписываться на получение событий, связанных с работой мыши, определять с их помощью, была ли нажата соответствующая клавиша, и только после этого вручную отображать само меню. Сейчас все стало проще: теперь управлением контекстным меню занимаются сами компоненты.

Сначала мы покажем пример того, как работать с компонентом `JPopupMenu` вручную. Следующее приложение, PopUpColorMenu, содер-

жит три кнопки. С помощью контекстного меню вы можете менять их цвет. Вы также можете выбрать фон для окна — все зависит от того, где был сделан щелок мышью.

```
//файл: PopUpColorMenu.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class PopUpColorMenu implements ActionListener
{
   Component selectedComponent;
   public PopUpColorMenu() {
JFrame frame = new JFrame("PopUpColorMenu v1.0");
final JPopupMenu colorMenu = new JPopupMenu("Цвет");
colorMenu.add(makeMenuItem("Красный"));
colorMenu.add(makeMenuItem("Зеленый"));
colorMenu.add(makeMenuItem("Синий"));
MouseListener mouseListener = new MouseAdapter() {
   public void mousePressed(MouseEvent e) { checkPopup(e); }
   public void mouseClicked(MouseEvent e) { checkPopup(e); }
   public void mouseReleased(MouseEvent e) { checkPopup(e); }
   private void checkPopup(MouseEvent e) {
      if (e.isPopupTrigger()) {
         selectedComponent = e.getComponent();
         colorMenu.show(e.getComponent(), e.getX(), e.getY());
      }
   }
};
// с версии 5.0 является опциональным
Container content = frame.getContentPane();

content.setLayout(new FlowLayout());
JButton button = new JButton("Один");
button.addMouseListener(mouseListener);
content.add(button);
button = new JButton("Два");
button.addMouseListener(mouseListener);
content.add(button);
button = new JButton("Три");
button.addMouseListener(mouseListener);
content.add(button);
```

```
        frame.getContentPane().addMouseListener(mouseListener);
        frame.setSize(200,50);
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        String color = e.getActionCommand();
        if (color.equals("Красный"))
            selectedComponent.setBackground(Color.red);
        else if (color.equals("Зеленый"))
            selectedComponent.setBackground(Color.green);
        else if (color.equals("Синий"))
            selectedComponent.setBackground(Color.blue);
    }
    private JMenuItem makeMenuItem(String label) {
        JMenuItem item = new JMenuItem(label);
        item.addActionListener( this );
        return item;
    }
    public static void main(String[] args) {
        new PopUpColorMenu();
    }
}
```

На рис. 17.8 показано, как это приложение выглядит после запуска; пользователь решил изменить цвет крайней правой кнопки.

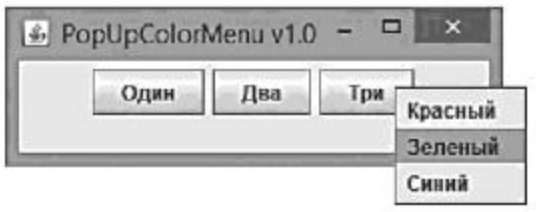

Рис. 17.8. Приложение PopupColorMenu

В этом примере контекстное меню можно вызывать из трех кнопок и главной панели окна; и так как делается это при помощи мыши, для каждого из этих компонентов нужно вручную зарегистрировать реализацию интерфейса `MouseListener`. В каждом случае используется один и тот же экземпляр анонимного внутреннего класса `MouseAdapter`. В этом классе мы переопределили методы `mousePressed()`, `mouseReleased()` и `mouseClicked()`, чтобы при получении подходящего события выводить контекстное меню. Но откуда мы знаем, какое событие является

«подходящим»? В этом нам помогает метод `isPopupTrigger()`, который скрывает от нас особенности реализации разных платформ. Если он возвращает `true`, мы знаем, что пользователь выполнил действие, которое в текущей системе должно вызывать контекстное меню.

Узнав об этом действии пользователя, мы отображаем компонент `JPopupMenu` с помощью метода `show()`, который в качестве аргументов принимает координаты указателя мыши из события `MouseEvent`.

Если бы мы хотели предоставлять разные меню для каждого компонента, нам бы пришлось создавать для этого отдельные реализации `MouseListener` со своими уникальными экземплярами `JPopupMenu`.

Теперь осталось только обеспечить обработку событий, возникающих в результате выбора пунктов меню. С помощью вспомогательного метода `makeMenuItem()` мы регистрируем окно `PopUpColorMenu` в качестве слушателя событий типа `ActionEvent` для каждого добавленного нами пункта. В нашем примере реализация `ActionListener` содержит метод `actionPerformed()`, который необходим для извлечения команд действия из события; по умолчанию эти команды совпадают с названиями пунктов. Затем в том же методе мы устанавливаем соответствующий цвет фона для выбранного компонента.

Управление контекстным меню на уровне отдельных компонентов

В платформе Java 5.0 появился новый API-интерфейс, который упрощает вызов контекстного меню из компонентов. Теперь любой объект типа `JComponent` может сам управлять экземпляром `JPopupMenu`, используя метод `setComponentPopupMenu()`. Кроме того, с помощью метода `setInheritsPopupMenu()` можно сделать так, чтобы компонент наследовал контекстное меню своего контейнера. Это позволяет легко реализовать объект `JPopupMenu`, который можно будет вызывать из любой части контейнера, независимо от компонента.

К сожалению, этот API-интерфейс не совсем подходит для нашего предыдущего примера. Во-первых, он не предоставляет информации о том, над каким компонентом находился указатель мыши в момент вызова контекстного меню.

Обычно обработка этого события не наследуется, а делегируется непосредственно контейнеру. Во-вторых, не все компоненты по умолча-

нию подписаны на получение событий типа MouseEvent.* Поэтому мы создадим новое приложение с более универсальным контекстным меню. В следующем примере под названием ContextMenu показаны компоненты TextArea и TextField, которые наследуют от своего контейнера JPanel один и тот же экземпляр JPopupMenu. Действие, выбранное с помощью пункта меню, отображается в многострочном поле ввода.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ContextMenu implements ActionListener
{
   JTextArea textArea = new JTextArea();
   public ContextMenu()
   {
      final JPopupMenu contextMenu = new JPopupMenu("Правка");
      contextMenu.add(makeMenuItem("Сохранить"));
      contextMenu.add(makeMenuItem("Сохранить как..."));
      contextMenu.add(makeMenuItem("Закрыть"));
      JFrame frame = new JFrame("ContextMenu v1.0");
      JPanel panel = new JPanel();
      panel.setLayout( new BorderLayout() );
      frame.getContentPane( ).add( panel );
      panel.setComponentPopupMenu( contextMenu );
      textArea.setInheritsPopupMenu( true );
      panel.add( BorderLayout.CENTER, textArea );
      JTextField textField = new JTextField();
      textField.setInheritsPopupMenu( true );
      panel.add( BorderLayout.SOUTH, textField );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(400,200);
      frame.setVisible(true);
   }
   public void actionPerformed( ActionEvent e ) {
      textArea.append( e.getActionCommand() +"\n" );
```

* Такие компоненты, как JPanel или JLabel, по умолчанию не занимаются обработкой событий, связанных с активностью мыши. Слушатель типа MouseListener сам подписывается на события MouseEvent. К сожалению, на момент написания этой книги подобного эффекта нельзя достичь с помощью метода setInheritsPopupMenu(). Чтобы обойти это ограничение, вы можете зарегистрировать для своих компонентов шаблонную реализацию MouseListener; это заставит их отслеживать события, связанные с мышью и, как следствие, позволит корректно вызывать контекстное меню.

```
    }
    private JMenuItem makeMenuItem(String label) {
        JMenuItem item = new JMenuItem(label);
        item.addActionListener( this );
        return item;
    }
    public static void main(String[] args) {
        new ContextMenu();
    }
}
```

Мы создали наш объект `JPopupMenu` так же, как и раньше, но на этот раз нам не нужно отслеживать нажатия клавиш мыши или вручную вызывать контекстное меню. Благодаря методу `setComponentPopupMenu()` всю эту работу за нас делает объект `JPanel`. Компоненты, из которых вызван метод `setInheritsPopupMenu()` (в нашем случае это `JTextArea` и `JTextField`), автоматически делегируют своему контейнеру события `MouseEvent`, приводящие к появлению контекстного меню.

Класс `JScrollPane`

Компонент `JScrollPane` уже использовался в этой главе, но мы так и не объяснили, что он собой представляет. Пришло время исправить это досадное упущение.

Объект `JScrollPane` — это контейнер, который может хранить другие компоненты. Если перефразировать, это *обертка* вокруг других компонентов. По умолчанию, если размеры хранящегося в нем объекта превышают его собственные, он выводит полосы прокрутки. События, связанные с этими полосами, контейнер `JScrollPane` обрабатывает самостоятельно, отображая соответствующую область содержащегося внутри компонента.

Объект `JScrollPane` выделяется на фоне других контейнеров. Его диспетчер компоновки (который, кстати, нельзя поменять) способен вместить не больше одного компонента. Но это ограничение легко обойти. Вы можете добавить все необходимые компоненты в контейнер `JPanel`, используя любой подходящий диспетчер компоновки, а полученный результат передать объекту `JScrollPane`.

При создании контейнера `JScrollPane` необходимо указать условия, при которых он должен отображать вертикальные и горизонтальные по-

лосы прокрутки. Для этого можно использовать следующие константы (отдельно для каждой из полос):

```
HORIZONTAL_SCROLLBAR_AS_NEEDED
VERTICAL_SCROLLBAR_AS_NEEDED
```

Полоса прокрутки выводится только в том случае, если компонент не помещается в контейнер.

```
HORIZONTAL_SCROLLBAR_ALWAYS
VERTICAL_SCROLLBAR_ALWAYS
```

Полоса прокрутки отображается всегда, независимо от размеров компонента, содержащегося в контейнере.

```
HORIZONTAL_SCROLLBAR_NEVER
VERTICAL_SCROLLBAR_NEVER
```

Полоса прокрутки никогда не появляется на экране, даже если компонент не помещается в контейнер. В случае использования этого условия вы должны предусмотреть какой-нибудь другой способ управления объектом `JScrollPane`.

По умолчанию действуют условия `HORIZONTAL_SCROLLBAR_AS_NEEDED` и `VERTICAL_SCROLLBAR_AS_NEEDED`.

Начиная с версии 1.4 платформа Java автоматически поддерживает прокрутку при помощи колеса мыши. Для этого вам не нужно изменять свой код. В следующем примере объект `JScrollPane` используется для вывода большого изображения (см. рис. 17.9). В коде этого примера нет ничего сложного; мы добавляем изображение в объект `JLabel`, который, в свою очередь, помещаем внутрь контейнера `JScrollPane`, а полученный результат выводим на главной панели в окне `JFrame`.



Рис. 17.9. Приложение `ScrollPaneFrame`

Ниже представлен код:

```
//файл: ScrollPaneFrame.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ScrollPaneFrame
{
   public static void main(String[] args) {
      String filename = "Piazza di Spagna.jpg";
      if (args.length > 0)
      filename = args[0];
      JFrame frame = new JFrame("ScrollPaneFrame v1.0");
      JLabel image = new JLabel( new ImageIcon(filename) );
      frame.getContentPane().add( new JScrollPane(image) );
      frame.setSize(300, 300);
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setVisible(true);
   }
}
```

Для вывода изображения на экран используются компоненты `JLabel` и `ImageIcon`. Класс `ImageIcon` загружает графический файл, и с помощью объекта `MediaTracker` определяет его размеры. Мы могли бы выводить изображение постепенно, проверяя состояние его загрузки. Управление графическими файлами будет рассмотрено в главе 21.

Класс JSplitPane

Разделяемая панель (`JSplitPane`) — это специальный контейнер, разбитый на две отдельные области, каждая из которых хранит свой компонент. Размеры областей можно менять. Чтобы было наглядней, представьте себе приложение для просмотра файлов, которое рядом с содержимым документа выводит его оглавление. Это и есть классический пример использования разделяемой панели.

Давайте возьмем предыдущее приложение и добавим в него еще один компонент `JLabel` с объектом `ImageIcon` внутри. Полученные изображения разместим на панели `JSplitPane`, используя контейнеры `JScrollPane` (см. рис. 17.10). Размеры компонентов можно изменять, перетаскивая разделитель в разные стороны.

```
//файл: SplitPaneFrame.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class SplitPaneFrame {
   public static void main(String[] args) {
      String fileOne = "Piazza di Spagna.jpg";
      String fileTwo = "L1-Light.jpg";
      if (args.length > 0) fileOne = args[0];
      if (args.length > 1) fileTwo = args[1];
      JFrame frame = new JFrame("SplitPaneFrame");
      JLabel leftImage = new JLabel( new ImageIcon( fileOne ) );
      Component left = new JScrollPane(leftImage);
      JLabel rightImage = new JLabel( new ImageIcon( fileTwo ) );
      Component right = new JScrollPane(rightImage);
      JSplitPane split =
         new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, left, right);
      split.setDividerLocation(100);
      frame.getContentPane().add(split);
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(300, 200);
      frame.setVisible(true);
   }
}
```

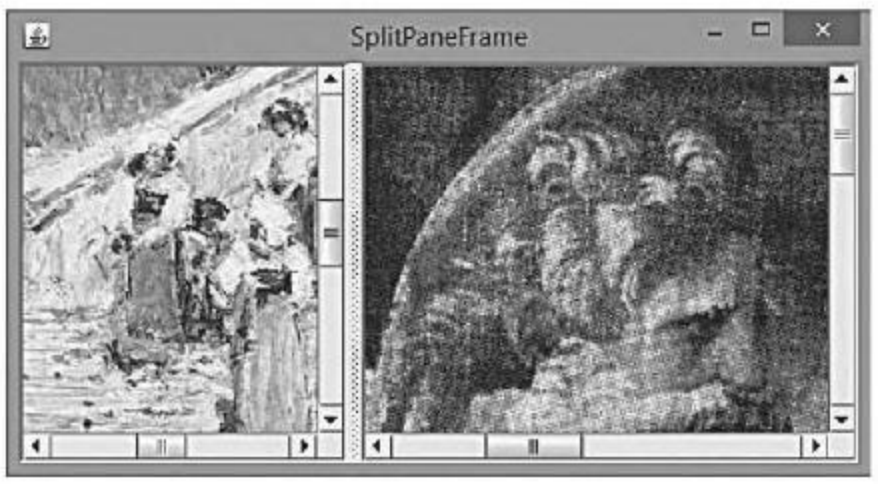

Рис. 17.10. Использование разделяемой панели

Класс JTabbedPane

Если вы когда-нибудь открывали контекстное меню своего рабочего стола, чтобы поменять настройки экрана (при условии, что вы пользователь операционной системы семейства Windows), у вас уже должно

быть представление о том, как выглядит компонент JTabbedPane. Это контейнер с именованными вкладками (такими как **Темы**, **Заставка** и **Оформление**, если говорить о настройках экрана). Если щелкнуть мышью по вкладке, компонент JTabbedPane отобразит новый набор элементов управления. В терминологии библиотеки Swing это всего лишь специальная разновидность контейнера. Новую вкладку можно добавить с помощью метода addTab(); при этом вам нужно будет указать ее имя и компонент, который представляет ее содержимое (обычно это контейнер, хранящий другие компоненты). Панель JTabbedPane может выводить одновременно только один набор компонентов, но имейте в виду, что при этом все остальные вкладки продолжают работать и занимать память. Если некоторые из ваших элементов управления активно потребляют ресурсы компьютера, вам стоит переводить их в «спящий» режим, когда они не отображаются на экране.

В следующем примере демонстрируется процесс создания объекта JTabbedPane. Мы добавим в него две вкладки: первая, **Компоненты**, будет содержать стандартные элементы управления библиотеки Swing, а для второй мы используем изображение с поддержкой прокрутки, которое было показано в предыдущих примерах.

```
//файл: TabbedPaneFrame.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class TabbedPaneFrame {
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("TabbedPaneFrame");
      JTabbedPane tabby = new JTabbedPane();
      // создаем панель с компонентами
      JPanel controls = new JPanel();
      controls.add(new JLabel("Сервис:"));
      JList list = new JList(
         new String[] { "Веб-сервер", "FTP-сервер" });
      list.setBorder(BorderFactory.createEtchedBorder());
      controls.add(list);
      controls.add(new JButton("Запуск"));
      // создаем панель с изображением
      String filename = "Piazza di Spagna.jpg";
```

```
        JLabel image = new JLabel( new ImageIcon(filename) );
        JComponent picture = new JScrollPane(image);
        tabby.addTab("Компоненты", controls);
        tabby.addTab("Изображение", picture);
        frame.getContentPane().add(tabby);
        frame.setSize(250, 200);
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible(true);
    }
}
```

Этот с виду непримечательный код создает довольно впечатляющий пользовательский интерфейс. В первой вкладке находится панель `JPanel`, которая содержит несколько других компонентов, включая список с гравированным обрамлением. Во вторую вкладку мы добавили контейнер `JScrollPane` с меткой `JLabel`, основанной на объекте `ImageIcon`. После запуска это приложение должно выглядеть так, как на рис. 17.11.

Наша панель содержит всего две вкладки, поэтому она довольно легко помещается в окне, но в реальных условиях свободное место может очень быстро закончиться. По умолчанию, если вкладок становится слишком много, они переносятся на новую строку. Визуально это похоже на картотеку, что довольно неплохо сочетается с концепцией компонента `JTabbedPane`. Но есть одна проблема: чтобы вывести на передний план вкладку из нижней строки, нужно менять позиции всех остальных вкладок. Это сбивает с толку многих пользователей и нарушает принципы проектирования пользовательского интерфейса, согласно которым элементы управления должны оставаться на своем месте. Вы можете избежать этой проблемы, сконфигурировав объект `JTabbedPane` так, чтобы все его вкладки выводились в одной строке и могли прокручиваться с помощью мыши; для этого передайте в метод `setTabLayoutPolicy()` константу `SCROLL_TAB_LAYOUT`:

```
setTabLayoutPolicy( JTabbedPane.SCROLL_TAB_LAYOUT );
```

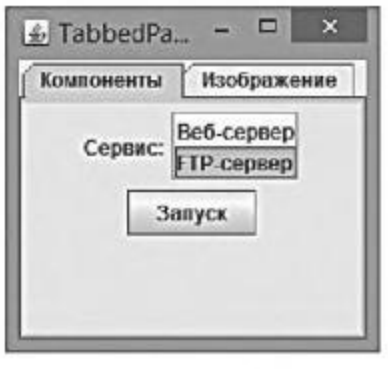

Рис. 17.11. Использование панели с вкладками

В платформе Java 6 появилась возможность размещать на вкладках дополнительные компоненты. Чаще всего это используется для добавления кнопок с различными функциями, такими как закрытие, сворачивание, вывод информации и т. д. В следующем примере мы добавим на каждую вкладку по одной кнопке, используя новый метод `setTabComponentAt(int index, Component component)`.

```
//файл: ClosableTabs.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class ClosableTabs extends JTabbedPane {
   public void addTab(String title, Color color) {
      JPanel pane = new JPanel();
      pane.setBackground(color);
      int loc = getTabCount();
      insertTab(title, null, pane, null, loc);
      setTabComponentAt(loc, new Tab(title));
   }
   public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable(){
         public void run(){
            JFrame frame = new JFrame("Закрывающиеся вкладки");
            ClosableTabs tabs = new ClosableTabs();
            tabs.addTab("Синяя", Color.BLUE);
            tabs.addTab("Зеленая", Color.GREEN);
            tabs.addTab("Красная", Color.RED);
            frame.add(tabs);
            frame.setDefaultCloseOperation(
               JFrame.EXIT_ON_CLOSE);
            frame.setSize(new Dimension(300, 150));
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);
         }});
   }
   /**
    * Компонент, использующийся в качестве вкладки (то есть,
    * сама вкладка, а не ее содержимое)
    */
   private class Tab extends JPanel {
      public Tab(String title) {
         super(new FlowLayout(FlowLayout.LEFT, 0, 0));
```

```
setOpaque(false);
// Название вкладки
JLabel label = new JLabel(title);
// Создаем отступ справа от кнопки закрытия
label.setBorder(BorderFactory.createEmptyBorder(0, 0,
   0, 2));
add(label);
// Кнопка для закрытия вкладки
JButton button = new CloseButton();
   add(button);
// Это нужно для вертикального выравнивания
// содержимого вкладки
setBorder(BorderFactory.createEmptyBorder(2, 0, 0, 0));
}
private class CloseButton extends JButton
   implements ActionListener {
   public CloseButton() {
      setPreferredSize(new Dimension(17, 17));
      setOpaque(false);
      setContentAreaFilled(false);
      setBorderPainted(false);
      setRolloverEnabled(true);
      setFocusable(false);
      addActionListener(this);
   }
   public void actionPerformed(ActionEvent e) {
      int i = ClosableTabs.this.indexOfTabComponent(
         Tab.this);
      if (i != -1) {
         ClosableTabs.this.remove(i);
      }
}
protected void paintComponent(Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   g2.setStroke(new BasicStroke(2));
   // Окрашиваем в красный цвет при наведении
   g2.setColor(Color.BLACK);
   if (getModel().isRollover()) {
   g2.setColor(Color.RED);
   }
   // Рисуем значок "X"
   int offset = 5;
```

```
            g2.drawLine(offset, offset, getWidth() -
               offset - 1, getHeight() - offset - 1);
            g2.drawLine(getWidth() - offset - 1, offset,
               offset, getHeight() - offset - 1);
            g2.dispose();
         }
      }
   }
}
```

Полосы прокрутки и ползунковые регуляторы

Контейнер `JScrollPane` настолько удобен, что в большинстве случаев использование отдельных полос прокрутки теряет всякий смысл. Но даже если они вам когда-нибудь понадобятся, имейте в виду, что для этих задач, скорее всего, лучше подойдет другой компонент — *ползунковый регулятор* (`JSlider`).

Чтобы не утомлять вас рассказами о внешнем виде и функциях этих компонентов, давайте сразу перейдем к примеру их практического применения. На рис. 17.12 показано приложение, которое выводит компоненты `JScrollBar` и `JSlider`.



Рис. 17.12. Использование полосы прокрутки и ползункового регулятора

Ниже представлен исходный код этого приложения:

```
//файл: Slippery.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class Slippery {
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Slippery v1.0");
```

```
// с версии 5.0 является опциональным
Container content = frame.getContentPane();
JPanel main = new JPanel(new GridLayout(2, 1));
JPanel scrollBarPanel = new JPanel();
final JScrollBar scrollBar =
   new JScrollBar(JScrollBar.HORIZONTAL, 0, 48, 0, 255);
int height = scrollBar.getPreferredSize().height;
scrollBar.setPreferredSize(new Dimension(175, height));
scrollBarPanel.add(scrollBar);
main.add(scrollBarPanel);
JPanel sliderPanel = new JPanel();
final JSlider slider =
   new JSlider(JSlider.HORIZONTAL, 0, 255, 128);
slider.setMajorTickSpacing(48);
slider.setMinorTickSpacing(16);
slider.setPaintTicks(true);
sliderPanel.add(slider);
main.add(sliderPanel);
content.add(main, BorderLayout.CENTER);
final JLabel statusLabel =
   new JLabel("Добро пожаловать в приложение Slippery v1.0");
content.add(statusLabel, BorderLayout.SOUTH);
// подключаем обработчики событий
scrollBar.addAdjustmentListener(new AdjustmentListener() {
   public void adjustmentValueChanged(AdjustmentEvent e) {
      statusLabel.setText("Текущее положение компонента JScrollBar =
"
         + scrollBar.getValue());
   }
});
slider.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
      statusLabel.setText("Текущее положение компонента JSlider = "
         + slider.getValue());
   }
});
frame.pack();
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setVisible(true);
   }
}
```

Здесь мы просто добавили в наше главное окно компоненты `JScrollBar` и `JSlider`. Информация об изменении их текущего положения отображается внизу с помощью метки `JLabel`.

При создании обоих этих компонентов необходимо указать их расположение — горизонтальное (`HORIZONTAL`) или вертикальное (`VERTICAL`). Можно также задать минимальное, максимальное и начальное значения. Объект `JScrollBar` поддерживает дополнительный параметр под названием `extent`; он определяет диапазон значений, который охватывает ползунок внутри полосы прокрутки. Например, если объекту `JScrollBar` с доступными значениями от 0 до 255 установить параметр `extent`, равный 128, то его ползунок будет занимать ровно половину прокручиваемой области.

Напротив некоторых значений компонент `JSlider` может выводить штриховые метки двух видов: *главные* (те, что потолще) и *второстепенные* (более мелкие). Для этого необходимо указать нужный интервал и вручную разрешить их отображение:

```
slider.setMajorTickSpacing(48);
slider.setMinorTickSpacing(16);
slider.setPaintTicks(true);
```

Рядом со штрихами можно выводить текстовые метки, используя метод `setLabelTable()`.

В обработке событий, генерируемых этими двумя компонентами, нет ничего сложного. При изменении положения объекты `JScrollBar` и `JSlider` передают события типа `AdjustmentEvent` и `ChangeEvent` соответственно. В вышеприведенном примере мы выводили новое значение измененного компонента с помощью метки `JLabel`, размещенной внизу окна.

Диалоговые окна

Еще одним стандартным элементом пользовательского интерфейса являются диалоговые окна. Они часто используются для вывода сообщений информационного («Ваш фруктовый салат готов») или вопросительного («Прикажете подать карету?») характера. Диалоговые окна являются настолько обыденной частью графических приложений, что для их создания в библиотеке Swing был предусмотрен целый ряд удобных компонентов. Доступ к ним можно получить с помощью ста-

тических методов класса `JOptionPane`. Существует множество разновидностей диалоговых окон; в классе `JOptionPane` они разбиты на четыре основные группы:

Сообщение

Выводят пользователю сообщение; обычно содержат кнопку **OK**.

Подтверждение

Задают вопрос, предлагая несколько вариантов ответа в виде кнопок (обычно это кнопки **Да**, **Нет** и **Отменить**).

Ввод

Просят пользователя ввести какой-то текст.

Выбор

Самый распространенный вид диалоговых окон, который позволяет выводить собственные компоненты.

На рис. 17.13 показано диалоговое окно, которое запрашивает подтверждение.

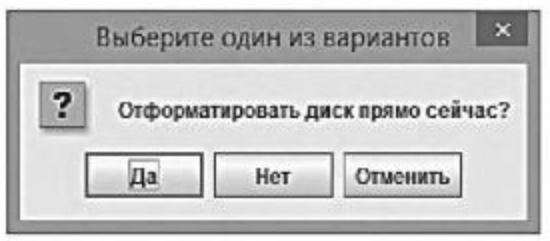

Рис. 17.13. Использование диалогового окна с подтверждением

Давайте рассмотрим каждое из этих диалоговых окон. Следующий код выводит сообщение:

```
JOptionPane.showMessageDialog(frame, "Пришло письмо.");
```

В качестве первого параметра для метода `showMessageDialog()` выступает родительский компонент (в нашем случае это переменная `frame` — окно приложения), по центру которого будет выводиться диалоговое окно. Если вы укажите **`null`**, сообщение появится в центре экрана. Окна, которые выводит класс `JOptionPane`, являются *модальными*; это означает, что пока они отображаются на экране, вы не можете работать с остальной частью приложения.

Ниже показан более интересный пример. Здесь помимо заголовка мы указываем тип сообщения, чтобы изменить значок диалогового окна:

```
JOptionPane.showMessageDialog(frame,
   "У вас заканчивается память.",
   "Апокалиптическое сообщение",
   JOptionPane.WARNING_MESSAGE);
```

Вот как выглядит код диалогового кона, представленного на рис. 17.13:

```
int result = JOptionPane.showConfirmDialog(null,
   "Отформатировать диск прямо сейчас?",
   "Выберите один из вариантов",
   JOptionPane.YES_NO_CANCEL_OPTION);
```

Чтобы сообщение появилось в центре экрана, мы указали **null** вместо родительского компонента. Метод `showConfirmDialog()` возвращает специальный код, сигнализируя о том, какая именно кнопка была нажата. Подробней об интерпретации этого кода мы расскажем позже в этом разделе.

Иногда у пользователя необходимо запросить ввод каких-то данных. Следующий код выводит диалоговое окно с просьбой ввести имя:

```
String name = JOptionPane.showInputDialog(null,
   "Пожалуйста, введите ваше имя.");
```

Любые введенные данные будут возвращены в виде объекта `String`; если пользователь нажмет кнопку **Отменить**, метод `showInputDialog()` вернет **null**.

Самый распространенный вид диалоговых окон можно создавать с помощью метода `showOptionDialog()`. Для этого вам нужно передать массив объектов, которые необходимо отобразить, а класс `JOptionPane` сам позаботится об их компоновке. Код, представленный ниже, выводит на экран следующие компоненты: `JLabel`, `JTextField` и `JPasswordField` (о компонентах для работы с текстом мы поговорим в следующей главе).

```
JTextField userField = new JTextField();
JPasswordField passField = new JPasswordField();
String message = "Пожалуйста, введите ваши логин и пароль.";
result = JOptionPane.showOptionDialog(frame,
   new Object[] { message, userField, passField },
   "Вход в систему", JOptionPane.OK_CANCEL_OPTION,
   JOptionPane.QUESTION_MESSAGE,
   null, null, null);
```

С помощью метода `showOptionDialog()` мы также указали заголовок («Вход в систему»). Чтобы вывести кнопки **OK** и **Отменить**, мы передали константу `OK_CANCEL_OPTION`, которая указывает на тип диалогового окна. Аргумент `QUESTION_MESSAGE` сигнализирует о том, что мы хотим отобразить значок с вопросительным знаком. Три заключительных параметра являются необязательными; с их помощью можно указать значок окна, массив различных вариантов и элемент, который будет выбран по умолчанию. В нашем случае будет использоваться стандартный значок. Если бы мы передали реальные значения для двух последних аргументов, класс `JOptionPane` мог бы отобразить указанные варианты в виде простого или раскрывающегося списка.

Следующее приложение включает в себя код из всех трех рассмотренных нами примеров:

```
import javax.swing.*;
public class ExerciseOptions {
   public static void main(String[] args) {
      JFrame frame = new JFrame("ExerciseOptions v1.0");
      frame.setSize(200, 200);
      frame.setVisible(true);
      JOptionPane.showMessageDialog(frame, "Пришло письмо.");
      JOptionPane.showMessageDialog(frame, "У вас заканчивается память.",
         "Апокалиптическое сообщение", JOptionPane.WARNING_MESSAGE);
      int result = JOptionPane.showConfirmDialog(null,
         "Отформатировать диск прямо сейчас?");
      switch (result) {
         case JOptionPane.YES_OPTION:
            System.out.println("Да"); break;
         case JOptionPane.NO_OPTION:
            System.out.println("Нет"); break;
         case JOptionPane.CANCEL_OPTION:
            System.out.println("Отменить"); break;
         case JOptionPane.CLOSED_OPTION:
            System.out.println("Закрыть"); break;
      }
      String name = JOptionPane.showInputDialog(null,
         "Пожалуйста, введите ваше имя.");
      System.out.println(name);
      JTextField userField = new JTextField();
      JPasswordField passField = new JPasswordField();
      String message = "Пожалуйста, введите ваши логин и пароль.";
```

```
        result = JOptionPane.showOptionDialog(frame,
            new Object[] { message, userField, passField },
            "Войти в систему", JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null, null, null);
        if (result == JOptionPane.OK_OPTION)
            System.out.println(userField.getText() +
                " " + new String(passField.getPassword()));
        System.exit(0);
    }
}
```

Диалоговое окно выбора файлов

Класс `JFileChooser` позволяет создавать стандартные диалоговые окна для выбора файлов. Как и любой другой компонент в библиотеке Swing, он полностью написан на языке Java, поэтому его внешний вид и поведение будут одинаковыми на всех платформах (хотя определенные визуальные стили позволяют использовать стандартное оформление текущей операционной системы). Выбор файлов сам по себе — довольно скучное занятие, поэтому мы придадим ему некоторый смысл. Давайте напишем небольшое приложение для редактирования текста, с помощью которого можно будет открывать и сохранять текстовые файлы (компонент `JFileChooser`, который будет выводиться в этом приложении, показан на рис. 17.14).

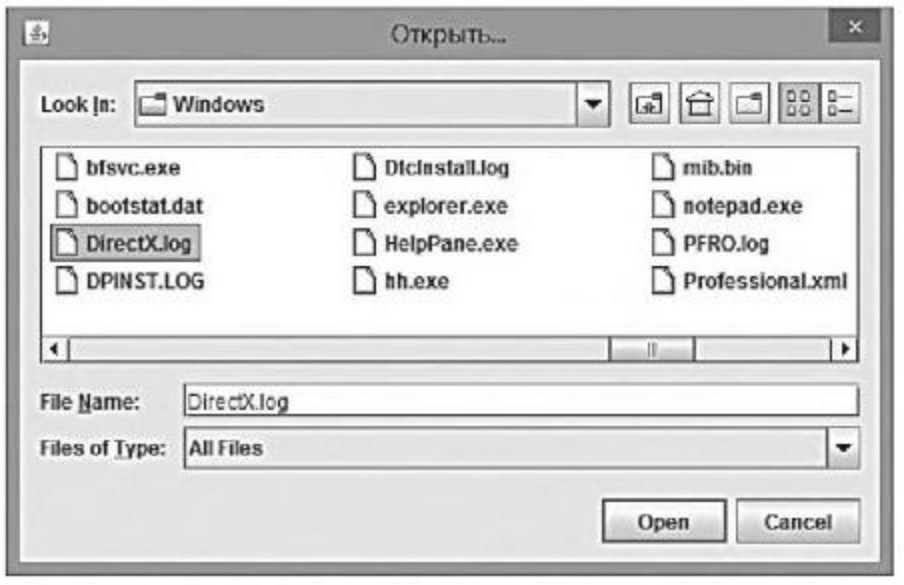

Рис. 17.14. Использование компонента `JFileChooser`

Для редактирования текста воспользуемся многострочным полем ввода (`JEditorPane`). Мы не стали реализовывать возможность сохранения файлов; предлагаем вам сделать это самостоятельно (хотя мы, все

же, поможем вам несколькими советами). Ниже представлен код приложения:

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
public class Editor extends JFrame implements ActionListener
{
   private JEditorPane textPane = new JEditorPane();
   public Editor() {
      super("Текстовый редактор");
      // с версии 5.0 является опциональным
      Container content = getContentPane();
      content.add(new JScrollPane(textPane), BorderLayout.CENTER);
      JMenu menu = new JMenu("Файл");
      menu.add(makeMenuItem("Открыть"));
      menu.add(makeMenuItem("Сохранить"));
      menu.add(makeMenuItem("Выйти"));
      JMenuBar menuBar = new JMenuBar();
      menuBar.add(menu);
      setJMenuBar(menuBar);
      setSize(300, 300);
      setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   }
   public void actionPerformed(ActionEvent e) {
      String command = e.getActionCommand();
      if (command.equals("Выйти")) System.exit(0);
      else if (command.equals("Открыть")) loadFile();
      else if (command.equals("Сохранить")) saveFile();
   }
   private void loadFile () {
      JFileChooser chooser = new JFileChooser();
      chooser.setDialogTitle("Открыть...");
      int result = chooser.showOpenDialog(this);
      if (result == JFileChooser.CANCEL_OPTION) return;
      try {
         File file = chooser.getSelectedFile();
         java.net.URL url = file.toURL();
         textPane.setPage(url);
      }
```

```
        catch (Exception e) {
            textPane.setText("Не удалось загрузить файл: " + e);
        }
    }
    private void saveFile() {
        JFileChooser chooser = new JFileChooser();
        chooser.showSaveDialog(this);
        // Сохраняем содержимое в файл...
    }
    private JMenuItem makeMenuItem( String name ) {
        JMenuItem m = new JMenuItem( name );
        m.addActionListener( this );
        return m;
    }
    public static void main(String[] s) {
        new Editor().setVisible(true);
    }
}
```

Класс `Editor` представляет собой окно типа `JFrame`, все пространство которого заполнено компонентом `JEditorPane` (о нем пойдет речь в главе 18) и строкой меню. Меню **Файл** содержит команды **Открыть**, **Сохранить** и **Выйти**. Метод `actionPerformed()` перехватывает события, связанные с этими командами, и выполняет соответствующие действия.

Интересным аспектом класса `Editor` является его частный метод `loadFile()`, который создает новый экземпляр `JFileChooser` и делает из него вызов `showOpenDialog()`. В результате появляется диалоговое окно для выбора файлов. Пока оно не завершит свою работу и не закроется, вызывающий поток будет заблокирован. После этого мы можем воспользоваться методом `getFile()`, чтобы получить выбранный файл. В методе `loadFile()` мы превращаем полученный экземпляр класса `File` в объект типа `URL` и передаем результат компоненту `JEditorPane`, который, в свою очередь, выводит содержимое выбранного файла. Компонент `JEditorPane` способен отображать документы в форматах HTML и RTF, в чем вы сможете убедиться в следующей главе.

Если вы хотите сами реализовать незаконченный метод `saveFile()`, вам стоит подумать о некоторых мерах предосторожности. Например, перед тем как перезаписывать файл, пользователю можно вывести одно из диалоговых окон для подтверждения рассмотренных выше, чтобы удостовериться в правильности сделанного выбора.

Диалоговое окно выбора цвета

Библиотека Swing под завязку набита всякими полезностями. Например, объект `JColorChooser` (еще одно готовое диалоговое окно) дает возможность выбирать цвета. Ниже представлено небольшое приложение, которое демонстрирует, насколько легко пользоваться этим компонентом:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class LocalColor {
   public static void main(String[] args) {
      final JFrame frame = new JFrame("Выбор цвета");
      // с версии 5.0 является опциональным
      final Container content = frame.getContentPane();

      content.setLayout(new GridBagLayout());
      JButton button = new JButton("Изменить цвет...");
      content.add(button);
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Color c = JColorChooser.showDialog(frame,
               "Выберите цвет", content.getBackground());
            if (c != null) content.setBackground(c);
         }
      });
      frame.setSize(200, 200);
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setVisible(true);
   }
}
```

Этот код выводит на экран одну кнопку, размещенную внутри контейнера `JFrame`. Когда вы нажмете эту кнопку, перед вами появится диалоговое окно, в котором можно будет выбрать цвет фона для главного окна приложения.

В сущности, данное приложение всего лишь вызывает из класса `JColorChooser` статический метод `showDialog()`. В этом примере мы указали родительский компонент, заголовок диалогового окна и цвет по умолчанию, но необходимым является только первый из этих трех аргументов. В итоге мы получим либо цвет, выбранный пользователем, либо значение **`null`**, если была нажата кнопка **Отменить**.

Глава 18

ДРУГИЕ КОМПОНЕНТЫ БИБЛИОТЕКИ SWING

В предыдущей главе мы уже обсудили большинство компонентов библиотеки Swing, предназначенных для построения пользовательского интерфейса. Остальные будут рассмотрены здесь. Речь пойдет о древовидных списках, таблицах и компонентах для работы с текстом. Несмотря на свои богатые возможности эти элементы управления довольно легко использовать в их стандартной конфигурации. Мы начнем со значений по умолчанию, постепенно переходя к более продвинутым возможностям. Позже в этой главе мы рассмотрим пример того, каким образом можно создать свой собственный компонент для библиотеки Swing.

Компоненты для работы с текстом

Библиотека Swing предлагает мощные средства для работы с текстом — от обычных полей ввода до компонентов, поддерживающих отображение HTML-страниц. Полноценный обзор всех этих инструментов можно найти в книге «Java Swing» издательства O'Reilly; это всеобъемлющее издание содержит несколько крупных глав, полностью посвященных работе с текстом. Мы же всего лишь пройдемся по основным аспектам этой огромной темы.

Давайте начнем с более простых компонентов. Однострочные и многострочные поля ввода представлены, соответственно, объектами `JTextField` и `JTextArea`. У них есть определенная общая функциональность, которую они наследуют от класса `JTextComponent`; это касается методов для установки и извлечения отображаемого текста, включения и отключения возможности редактирования, перемещения текстового курсора и работы с выделенными текстовыми фрагментами.

Чтобы отслеживать изменения, возникающие при редактировании текста, необходимо понимать, как компоненты реализуют архитектуру MVC (модель — представление — контроллер). Если помните, в предыдущей главе мы говорили, что библиотека Swing имеет полноценную

поддержку этой парадигмы. Как раз на примере компонентов для работы с текстом вы впервые увидите четкое разделение между моделью данных (M) и остальной частью приложения (VC). В качестве модели у нас будет выступать объект типа `Document`. На нем будут отражаться все изменения, происходящие с содержимым объектов `JTextField` и `JTextArea` (например, добавление или удаление текста). При редактировании текста события генерирует сам документ, а не визуальный элемент управления. Таким образом, чтобы получать уведомления об изменениях в компоненте `JTextArea`, нужно подписаться на его внутренний объект `Document`:

```
JTextArea textArea = new JTextArea();
Document doc = textArea.getDocument();
doc.addDocumentListener(someListener);
```

Как вы увидите в следующем примере, несколько визуальных компонентов вполне могут иметь одну общую модель данных.

Стоит также сказать, что при каждом нажатии клавиши **Enter** в рамках поля ввода объект `JTextField` генерирует события типа `ActionEvent`. Для получения этих событий нужно реализовать интерфейс `ActionListener` и зарегистрироваться с помощью метода `addActionListener()`.

В следующих разделах представлено несколько простых приложений, на примере которых вы узнаете, как работать с компонентами `JTextField` и `JTextArea`.

Приложение TextEntryBox

В первом примере под названием TextEntryBox мы свяжем между собой два компонента: `JTextArea` и `JTextField` (см. рис. 18.1).

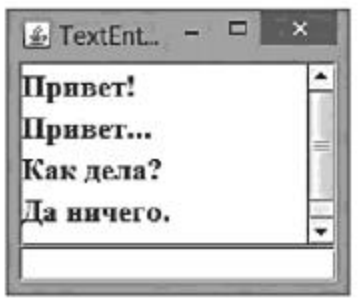

Рис. 18.1. Приложение TextEntryBox

Если пользователь во время набора текста в компоненте `JTextField` нажмет клавишу **Enter**, мы получаем событие `ActionEvent`; так мы узна-

ем, что в многострочное поле ввода `JTextArea` нужно добавить новую строчку. Попробуйте сами. Чтобы компонент `JTextField` смог принимать ввод данных, вам, возможно, придется выделить его при помощи щелчка мышью. Когда окно наполнится текстом, вы сможете увидеть полосу прокрутки:

```
//файл: TextEntryBox.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TextEntryBox {
   public static void main(String[] args) {
JFrame frame = new JFrame("TextEntryBox");
final JTextArea area = new JTextArea();
area.setFont(new Font("Serif", Font.BOLD, 18));
area.setText("Привет!\n");
final JTextField field = new JTextField();
frame.add(new JScrollPane(area), BorderLayout.CENTER);
frame.add(field, BorderLayout.SOUTH);
field.requestFocus();
field.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent ae) {
      area.append(field.getText() + '\n');
      field.setText("");
   }
});
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setSize(200, 300);
      frame.setVisible(true);
   }
}
```

Объект `TextEntryBox` слишком уж простой, поэтому мы попытались сделать его чуть более интересным. С помощью метода `setFont()` из класса `Component` мы увеличили размер шрифта (речь о шрифтах пойдет в главе 20). Кроме того, чтобы получать уведомления о нажатии клавиши **Enter**, мы зарегистрировали анонимный внутренний класс для отслеживания событий типа `ActionEvent`.

Нажатие клавиши **Enter** внутри поля ввода `JTextField` генерирует событие — вот где начинается все самое интересное. Обработка события

выполняется в методе `actionPerformed()` нашей внутренней реализации интерфейса `ActionListener`. После этого с помощью методов `getText()` и `setText()` мы получаем и передаем введенный пользователем текст. Эти методы принадлежат классу `JTextComponent`, который является родительским для объектов `JTextField` и `JTextArea`, поэтому их можно использовать в обоих полях ввода.

Обработчик событий, `actionPerformed()`, вызывает метод `field.getText()`, чтобы прочитать текст, содержащийся в компоненте `JTextField`. Сделав это, он добавляет полученный текст в многострочное поле ввода, используя метод `area.append()`. В конце, чтобы подготовить компонент `JTextField` для набора нового текста, мы очищаем его с помощью вызова `field.setText("")`.

Помните, что модель данных и компонент для работы с текстом — это разные вещи. Все вызовы методов `setText()`, `getText()` и `append()` делегируются исходному объекту `Document`.

По умолчанию вышеописанные компоненты доступны для ввода и редактирования. Чтобы разрешить исключительно вывод текста, можно сделать вызов `setEditable(false)`. Оба эти поля ввода дают возможность копировать, вырезать и вставлять *выделенные* фрагменты текста в рамках текущей оконной системы. Чтобы создать выделение, нужно нажать клавишу мыши над полем ввода и переместить указатель в нужном направлении. Передавать текст между разными окнами можно с помощью стандартных клавиатурных сочетаний; в большинстве систем это **Ctrl+C** (копирование), **Ctrl+V** (вставка) и **Ctrl+X** (вырезание). В OS X все то же самое, только вместо **Ctrl** нужно нажимать клавишу **Command**. Эти операции можно выполнять и программным способом, используя методы `cut()`, `copy()` и `paste()` из класса `JTextComponent`. Такая возможность может пригодиться при создании контекстного меню со стандартными командами для управления текстом. Получить текущий выделенный фрагмент можно с помощью метода `getSelectedText()`; для выделения части текста есть метод `selectText()`, который принимает диапазон индексных значений; чтобы выделить все содержимое поле ввода, используйте метод `selectAll()`.

Обратите внимание, что наш компонент `JTextArea` аккуратно размещен внутри контейнера `JScrollPane`. Таким образом, когда для отображения содержимого поля ввода не хватает свободного места, на экране появляются полосы прокрутки.

Форматированный текст

Компонент `JFormattedTextField` специально предназначен для работы со сложными форматированными значениями, такими как числа и даты. Он чем-то похож на обычное поле ввода `JTextField`, но его конструктор принимает специальные объекты, предназначенные для работы с определенными типами данных (такими как `Date` или `Integer`); сами данные передаются и принимаются с помощью методов `setValue()` и `getValue()`. В следующем примере демонстрируется процесс создания простой формы с полями для ввода разных видов форматированных значений:

```
import java.text.*;
import javax.swing.*;
import javax.swing.text.*;
import java.util.Date;
public class FormattedFields
{
   public static void main( String[] args ) throws Exception {
      Box form = Box.createVerticalBox();
      form.add( new JLabel("Имя:") );
      form.add( new JTextField("Иван Иванов") );
      form.add( new JLabel("Дата рождения:") );
      JFormattedTextField birthdayField =
      new JFormattedTextField(new SimpleDateFormat("MM/dd/yy"));
      birthdayField.setValue( new Date() );
      form.add( birthdayField );
      form.add( new JLabel("Возраст:") );
      form.add(new JFormattedTextField(new Integer(32)));
      form.add( new JLabel("Волосяной покров на теле:") );
      JFormattedTextField hairsField
         = new JFormattedTextField( new DecimalFormat("###,###") );
      hairsField.setValue(new Integer(100000));
      form.add( hairsField );
      form.add( new JLabel("Номер телефона:") );
      JFormattedTextField phoneField =
         new JFormattedTextField( new MaskFormatter("(###)###-####") );
      phoneField.setValue("(314)555-1212");
      form.add( phoneField );
      JFrame frame = new JFrame("Информация о пользователе");
      frame.getContentPane().add(form);
      frame.pack();
```

```
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible(true);
    }
}
```

Объект `JFormattedTextField` можно создавать разными способами. Вы можете использовать обычный экземпляр класса `java.lang.Number` (например, `Integer` или `Float`) в качестве прототипа или вручную описывать правила форматирования с помощью одного из объектов, предоставляемых пакетом `java.text`: `java.text.NumberFormat`, `java.text.DateFormat` или более универсального `java.text.MaskFormatter` (первые два из них мы уже обсуждали в главах 10 и 11). Класс `MaskFormatter` позволяет создавать произвольные правила форматирования. В следующих разделах мы поговорим о том, как фильтровать и проверять данные, которые вводятся с помощью визуальных компонентов; вы узнаете, как ограничивать ввод определенных символов и анализировать полученные значения. Напоследок стоит упомянуть, что в этом примере использовался контейнер `Box`, диспетчером компоновки для которого служит объект `BoxLayout` (о нем мы поговорим в главе 19).

Итак, чтобы передать текст компоненту `JFormattedTextField`, используйте метод `setValue()`; метод `getValue()` позволяет получить последнее корректное значение. Во втором случае вам придется привести полученный объект к подходящему типу, учитывая при этом заданный формат. Например, следующий код извлекает дату из поля `birthdayField`, которое мы использовали в вышеприведенном примере:

```
Date bday = (Date)birthdayField.getValue();
```

Содержимое объекта `JFormattedTextField` проверяется в тот момент, когда пользователь пытается перейти к другому полю ввода (с помощью щелчка мышью или посредством клавиатуры). По умолчанию реакцией на ввод любых недопустимых данных является возврат к последнему корректному значению. Если вы хотите, чтобы текст оставался в поле ввода для дальнейшего редактирования, можете передать в метод `setFocusLostBehavior()` константу `JFormattedTextField.COMMIT` (изначально используется `COMMIT_OR_REVERT`). Как бы то ни было, некорректный ввод не влияет на итоговое значение, которое извлекается с помощью метода `getValue()`.

Фильтрация ввода

Сам по себе класс `JFormattedTextField` не знает ни о каких форматах; эти сведения он получает из объектов типа `AbstractFormatter`,

которые являются реализацией двух интерфейсов: `DocumentFilter` и `NavigationFilter`. Первый из них, `DocumentFilter`, позволяет подключаться к реализациям модели `Document` и перехватывать команды редактирования, изменяя их по своему усмотрению. Второй, `NavigationFilter`, используется для подключения к компонентам `JTextComponent`, чтобы управлять перемещениями текстового курсора (так же, как в полях ввода с объектом форматирования `MaskFormatter`). В компоненте `JFormattedTextField` можно использовать собственную реализацию `AbstractFormatter`, а интерфейс `DocumentFilter` позволяет управлять редактированием документов и вводом текста. Например, с его помощью вы могли бы переводить набираемые пользователем символы в верхний регистр или вставлять вместо них какие-нибудь иероглифы. Это низкоуровневый инструмент, который позволяет управлять вводом на лету. Пример его использования представлен в следующем подразделе. С помощью подхода, который применялся при форматировании текста, мы попробуем реализовать для нашего поля ввода высокоуровневую проверку содержимого, которая будет обеспечивать корректность вводимых данных.

Класс `DocumentFilter`

Следующее приложение под названием DocFilter применяет реализацию интерфейса `DocumentFilter` к компоненту `JTextField`. Наш фильтр будет просто переводить любой текст в верхний регистр. Ниже представлен код:

```
import java.text.*;
import javax.swing.*;
import javax.swing.text.*;
public class DocFilter
{
   public static void main( String[] args ) throws Exception
   {
      JTextField field = new JTextField(30);
      ((AbstractDocument)(field.getDocument())).setDocumentFilter(
         new DocumentFilter()
      {
         public void insertString(
            FilterBypass fb, int offset, String string, AttributeSet attr)
            throws BadLocationException
         {
```

```
            fb.insertString( offset, string.toUpperCase(), attr );
        }
        public void replace(
            FilterBypass fb, int offset, int length, String string,
            AttributeSet attr) throws BadLocationException
        {
            fb.replace( offset, length, string.toUpperCase(), attr );
        }
    } );
    JFrame frame = new JFrame("Информация о пользователе");
    frame.add( field );
    frame.pack();
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible(true);
    }
}
```

Методы `insertString()` и `replace()` из класса `DocumentFilter` вызываются при добавлении нового или изменении существующего текста. С их помощью мы можем фильтровать введенные значения перед тем, как передавать их дальше. Когда текст готов, мы вставляем его в документ с помощью ссылки типа `FilterBypass`. Объект `FilterBypass` содержит несколько методов, которые позволяют применять изменения непосредственно к документу. Метод `remove()` из класса `DocumentFilter` может использоваться для перехватывания правок, связанных с удалением символов из текста.

Стоит также отметить, что не все реализации интерфейса `Document` содержат метод `setDocumentFilter()`. В таких случаях приходится приводить объект к типу `AbstractDocument`, ведь только документы, наследованные от этого класса, могут принимать фильтры (как вариант, вы можете сами реализовать механизм фильтрации). Такое печальное положение дел обусловлено тем, что поддержка технологии Document Filter API появилась на платформе Java только в версии 1.4, и при этом было решено, что старый интерфейс `Document` должен остаться неизменным.

Проверка данных

Низкоуровневая фильтрация ввода позволяет предотвратить ситуации, когда вы ожидаете получить что-то одно (например, букву), а вам передают совсем другое (например, число). В этом разделе мы погово-

рим о высокоуровневой проверке данных, благодаря которой мы сможем учитывать, что в феврале 28 дней, понимать разницу между номерами кредитных карт Visa и MasterCard и т. д. Если фильтрация символов не дает пользователю ввести некорректную информацию, то проверка полей проводится уже после того, как текст был введен. Обычно это происходит при попытке перехода к другому элементу управления (с помощью мыши или клавиатуры). В платформе Java 1.4 появился класс `InputVerifier`, который позволяет проверять содержимое компонента до того, как произойдет передача фокуса. И хотя сейчас мы рассматриваем компоненты, предназначенные для работы с текстом, с помощью класса `InputVerifier` можно проверять состояние любого объекта типа `JComponent`.

Код, представленный ниже, создает два поля ввода. Первое принимает любые значения, а второе допускает ввод только чисел в диапазоне от 0 до 100. Переключаться между ними можно только в том случае, если введенные данные отвечают установленным критериям. Но если второе поле будет содержать некорректное значение, при попытке перехода к другому компоненту программа издаст звуковой сигнал и выделит набранный текст. Фокус нельзя будет сместить в другое место до тех пор, пока вы не устраните проблему.

```
import javax.swing.*;
public class Validator
{
   public static void main( String[] args ) throws Exception {
      Box form = Box.createVerticalBox();
      form.add( new JLabel("Любое начение") );
      form.add( new JTextField("5000") );
      form.add( new JLabel("Только от 0 до 100") );
      JTextField rangeField = new JTextField("50");
      rangeField.setInputVerifier( new InputVerifier() {
         public boolean verify( JComponent comp ) {
            JTextField field = (JTextField)comp;
            boolean passed = false;
            try {
               int n = Integer.parseInt(field.getText());
               passed = ( 0 <= n && n <= 100 );
            } catch (NumberFormatException e) { }
            if ( !passed ) {
               comp.getToolkit().beep();
```

```
            field.selectAll();
        }
        return passed;
    }
} );
form.add( rangeField );
JFrame frame = new JFrame("Информация о пользователе");
frame.add(form);
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.pack();
frame.setVisible(true);
}
```

В этом коде мы создали внутреннюю анонимную производную класса `InputVerifier`. Алгоритм крайне прост; во время проверки данных вызывается наш метод `verify()`, которому мы передаем ссылку на компонент, чье содержимое нужно проанализировать. Мы приводим данные к подходящему типу (конечно же, мы знаем, что проверяем) и смотрим, находится ли число в нужном диапазоне. Если нет — издаем звуковой сигнал и выделяем текст. В конце с помощью значений `true` или `false` мы сигнализируем о том, прошло ли значение нашу проверку.

Вы можете использовать класс `InputVerifier` в сочетании с компонентом `JFormattedTextField`, чтобы помимо соблюдения корректного формата обеспечить проверку данных, которые набирает пользователь.

Сезам, откройся!

Прежде чем заканчивать разговор о форматированном тексте, следует сказать, что библиотека Swing включает в себя компонент `JPasswordField`, который специально предназначен для ввода паролей. Он является наследником класса `JTextField`, и ведет себя похожим образом, но любой текст, который в него вводится, отображается в виде одинаковых малоинформативных символов (как правило, звездочек). На рис. 18.2 показано диалоговое окно из предыдущей главы, в котором вместо нижнего поля ввода используется компонент `JPasswordField`.

В плане создания и использования объект `JPasswordField` ничем не отличается от своего родственника, `JTextField`. Есть только один нюанс: если вам не нравятся звездочки, с помощью метода `setEchoChar()` вы можете их заменить другим символом.

Обычно, чтобы получить текст, содержащийся в поле ввода `JPasswordField`, вызывается метод `getText()`. Но имейте в виду, что этот способ считается устаревшим; сейчас рекомендуется использовать метод `getPassword()`, который вместо объекта `String` возвращает массив символов. Это было сделано из соображений безопасности, так как пароль, хранящийся в виде массива символов, сложнее обнаружить при анализе оперативной памяти; к тому же это позволяет удалять содержимое поля ввода напрямую. Если эти вопросы вас не очень волнуют, вы можете просто создать из полученного значения новую строку. Стоит сказать, массив символов является стандартным представлением пароля на платформе Java, поэтому данные, возвращаемые методом `getPassword()`, можно сразу передавать в криптографические классы, не создавая промежуточной строки.

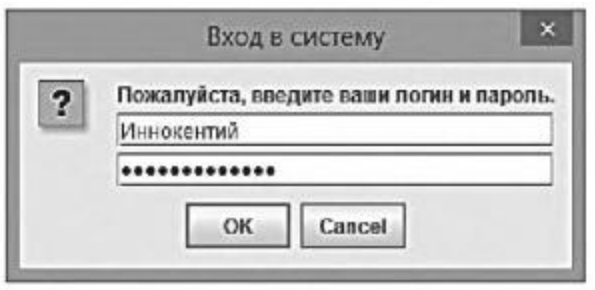

Рис. 18.2. Использование компонента `JPasswordField` в диалоговом окне

Использование общей модели данных

В нашем следующем примере показано, насколько легко можно назначить нескольким полям ввода одну и ту же реализацию интерфейса `Document`. Внешний вид приложения можно оценить на рис. 18.3.

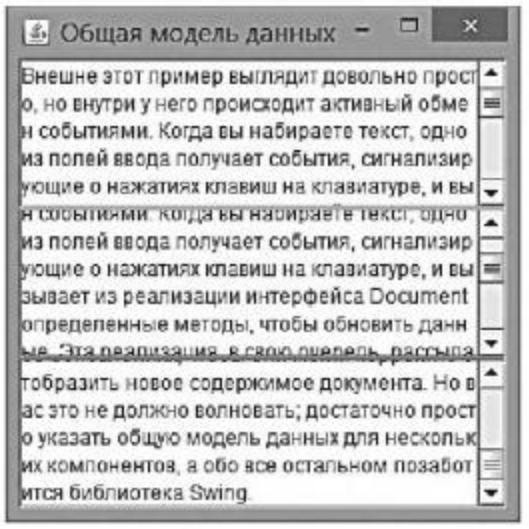

Рис. 18.3. Три представления одной и той же модели данных

Изменения, вносимые в содержимое любого из этих полей ввода, будут отражаться на остальных компонентах. Для этого нам просто нужно назначить им общую модель данных:

```
JTextArea areaFiftyOne = new JTextArea();
JTextArea areaFiftyTwo = new JTextArea();
areaFiftyTwo.setDocument(areaFiftyOne.getDocument());
JTextArea areaFiftyThree = new JTextArea();
areaFiftyThree.setDocument(areaFiftyOne.getDocument());
```

Точно так же мы могли бы назначить один документ сразу для семи (или семидесяти) компонентам `JTextArea`. Может быть, этот пример и не выглядит особенно полезным, но имейте в виду, что он позволяет одновременно просматривать разные участки одного и того же документа — это одна из прелестей использования нескольких представлений с общей моделью данных. Вы также можете выводить один набор данных разными способами. Например, числовую информацию можно представить одновременно в виде электронной таблицы и круговой диаграммы. Архитектура MVC, которая используется в библиотеке Spring, позволяет делать такие вещи «с умом»; то есть, если изменить данные в электронной таблице, то диаграмма, которая использует ту же модель, обновится автоматически.

Внешне этот пример выглядит довольно просто, но внутри у него происходит активный обмен событиями. Когда вы набираете текст, одно из полей ввода получает события, сигнализирующие о нажатиях клавиш на клавиатуре, и вызывает из реализации интерфейса `Document` определенные методы, чтобы обновить данные. Эта реализация, в свою очередь, рассылает события другим компонентам, сообщая им об обновлениях, чтобы они могли корректно отобразить новое содержимое документа. Но вас это не должно волновать; достаточно просто указать общую модель данных для нескольких компонентов, а обо все остальном позаботится библиотека Swing.

```
//файл: SharedModel.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class SharedModel {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Общая модель данных");
      JTextArea areaFiftyOne = new JTextArea();
      JTextArea areaFiftyTwo = new JTextArea();
      areaFiftyTwo.setDocument(areaFiftyOne.getDocument());
      JTextArea areaFiftyThree = new JTextArea();
      areaFiftyThree.setDocument(areaFiftyOne.getDocument());
```

```
        frame.setLayout(new GridLayout(3, 1));
        frame.add(new JScrollPane(areaFiftyOne));
        frame.add(new JScrollPane(areaFiftyTwo));
        frame.add(new JScrollPane(areaFiftyThree));
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
```

В выводе на экран всех этих компонентов нет ничего сложного. С помощью диспетчера компоновки `GridLayout` (о котором мы поговорим в следующей главе) мы добавляем в окно три многострочных поля ввода. После этого нам остается только назначить им общую реализацию интерфейса `Document`.

Встроенная поддержка форматов HTML и RTF

В большинстве пользовательских интерфейсов используется всего две производных класса `JTextComponent`. Речь идет о компонентах `JTextField` и `JTextArea`, которые мы только что рассмотрели. Но это только верхушка айсберга. Библиотека Swing предоставляет два сложных и мощных инструмента для работы с текстом, которые тоже унаследованы от класса `JTextComponent`: `JEditorPane` и `JTextPane`.

Первый из них, `JEditorPane`, способен отображать файлы в форматах HTML и RTF (Rich Text Format — формат обогащенного текста) и предоставляет систему расширений для поддержки других документов. Он может генерировать событие типа `HyperlinkEvent`, производные которого передаются во время появления и исчезновения курсора мыши над гиперссылкой, а также при щелчке по ней. Учитывая то, что компонент `JEditorPane` поддерживает отображение HTML-страниц, перед нами вырисовывается заманчивая возможность написать простой веб-браузер, и, конечно же, мы не преминем ею воспользоваться. На создание приложения, показанного на рис. 18.4, ушло всего 70 строчек кода.

```
//файл: CanisMinor.java
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;
```

```
import javax.swing.event.*;
public class CanisMinor extends JFrame {
   protected JEditorPane mEditorPane;
   protected JTextField mURLField;
   public CanisMinor(String urlString) {
      super("CanisMinor v1.0");
      createGUI(urlString);
   }
   protected void createGUI( String urlString ) {
      setLayout(new BorderLayout());
      JToolBar urlToolBar = new JToolBar();
      mURLField = new JTextField(urlString, 40);
      urlToolBar.add(new JLabel("Адрес "));
      urlToolBar.add(mURLField);
      add(urlToolBar, BorderLayout.NORTH);
      mEditorPane = new JEditorPane();
      mEditorPane.setEditable(false);
      add(new JScrollPane(mEditorPane), BorderLayout.CENTER);
      openURL(urlString);
      mURLField.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {
            openURL(ae.getActionCommand());
         }
      });
      mEditorPane.addHyperlinkListener(new LinkActivator());
      setSize(500, 600);
      setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   }
   protected void openURL(String urlString) {
      try {
         URL url = new URL(urlString);
         mEditorPane.setPage(url);
         mURLField.setText(url.toExternalForm());
      }
      catch (Exception e) {
         System.out.println("Не удалось открыть " + urlString + ":" + e);
      }
   }
   class LinkActivator implements HyperlinkListener {
      public void hyperlinkUpdate(HyperlinkEvent he) {
         HyperlinkEvent.EventType type = he.getEventType();
```

```
            if (type == HyperlinkEvent.EventType.ACTIVATED)
               openURL(he.getURL().toExternalForm());
         }
      }
      public static void main(String[] args) {
         String urlString = "http://ru.wikinews.org/wiki/Special:Random";
         if (args.length > 0)
            urlString = args[0];
         new CanisMinor( urlString ).setVisible( true );
      }
}
```

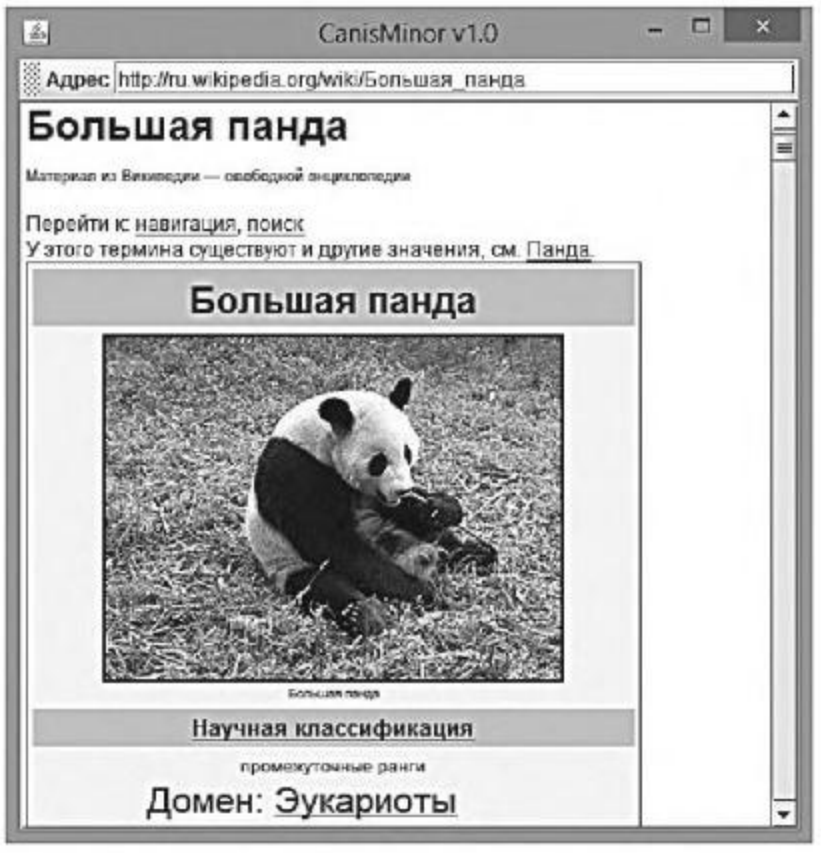

Рис. 18.4. Простой веб-браузер

В основе этого небольшого приложения лежит компонент `JEditorPane`. Чтобы загрузить новую страницу, мы передаем его методу `setPage()` объект `URL`, который указывает либо на локальный файл, либо на какой-то ресурс в Интернете. Но сначала нужно набрать адрес страницы в поле ввода, которое находится вверху, и нажать клавишу **Enter**. В результате генерируется событие `ActionEvent`, которое передает введенный адрес объекту `JEditorPane`. С помощью этого приложения мы также можем просматривать документы в формате RTF (этот формат используется для хранения текстовых и прочих недвоичных данных в таких программах как Microsoft Word).

Чтобы обеспечить адекватную реакцию на взаимодействие пользователя с гиперссылками, нам достаточно выполнять обработку событий

HyperlinkEvent, генерируемых компонентом JEditorPane. Вся эта обработка выполняется во внутреннем классе LinkActivator. Нас интересует только «активация» гиперссылки — это когда пользователь щелкает по ней мышью. В ответ на это действие мы передаем объекту JEditorPane соответствующий адрес. Вот, собственно, и все.

Внутри компонента JEditorPane находится объект EditorKit, который и занимается отображением документов. У этого объекта есть несколько разновидностей, предназначенных для работы с разными форматами данных. Например, для вывода HTML-страниц используется класс HTMLEditorKit (из пакета javax.swing.text.html).

На сегодняшний день поддерживается устаревший стандарт HTML 3.2, и, несмотря на обещания, спецификация HTML 4.0 не реализована даже в платформе Java 7. Класс HTMLEditorKit поддерживает некоторые привычные для нас возможности языка HTML — например, он автоматически отправляет содержимое веб-формы при нажатии соответствующей кнопки (хотя этот процесс можно переопределить с помощью события FormSubmitEvent).

Побродив немного по Интернету с помощью этого приложения, вы наверняка обнаружите, что класс HTMLEditorKit не может отобразить большинство современных веб-сайтов. В своем нынешнем состоянии он лучше всего подходит для простых задач, таких как просмотр справочных материалов. Если вам нужно что-то более функциональное, советуем обратить внимание на прекрасный коммерческий компонент WebRenderer от компании JadeLiquid.

Есть еще один элемент управления, о котором мы до сих пор не упоминали — панель инструментов (JToolBar). Как раз внутри этого многофункционального контейнера размещено наше поле для ввода URL-адресов.

Изначально наш экземпляр JToolBar находится в верхней части окна, но щелкнув мышью по пунктирной области в его левой части, вы можете переместить его куда угодно — влево, вправо или вниз. Более того, его можно полностью вывести за пределы главного контейнера, и тогда у него появится свое собственное окно. Это все делается автоматически на уровне класса JToolBar. Нам остается только создать панель инструментов и добавить в нее какие-нибудь элементы управления. Компонент JToolBar является обычным контейнером, поэтому изначально мы разместили его на главной панели нашего окна.

Работа с текстом вручную

У класса `JTextComponent` есть еще одна производная, которая позволяет делать с текстом все что угодно — `JTextPane`. В отличие от компонентов `JTextField` и `JTextArea`, она может использовать одновременно множество шрифтов и стилей начертания. Она также поддерживает выделение, вставку изображений и другие продвинутые возможности.

Чтобы разобраться с компонентом `JTextPane`, мы создадим многострочное поле ввода со стилизованным текстом. Помните, сам текст хранится внутри модели данных, объекте `Document`. Чтобы применить какой-либо стиль, нужно назначить определенной части документа набор атрибутов. Для работы с этими атрибутами библиотека Swing предоставляет специальные классы и методы; с их помощью вы сможете, к примеру, использовать полужирный шрифт или изменять цвет текста. Сами атрибуты находятся в классе `SimpleAttributeSet`, а статические методы для управления ими предоставляет класс `StyleConstants`. Давайте создадим стиль, который окрашивает текст в красный цвет:

```
SimpleAttributeSet redstyle = new SimpleAttributeSet();
StyleConstants.setForeground(redstyle, Color.red);
```

Чтобы добавить в документ красный текст, нужно просто передать в метод `insertString()` строку и атрибуты:

```
document.insertString(6, "Какой-нибудь красный текст", redstyle);
```

В качестве первого аргумента метод `insertString()` принимает количество символов, к которым будет применен стиль. Если переданное число превысит длину самого документа, будет сгенерировано исключение. Если вместо набора атрибутов указать значение `null`, компонент `JTextPane` применит шрифт и стиль по умолчанию.

В следующем примере мы создадим несколько наборов атрибутов, и используем их для добавления в поле ввода `JTextPane` обычного и стилизованного текста (см. рис. 18.5).

```
//файл: Styling.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;
public class Styling extends JFrame {
   private JTextPane textPane;
```

```
public Styling() {
   super("Стилизованный текст");
   setSize(300, 200);
   textPane = new JTextPane();
   textPane.setFont(new Font("Serif", Font.PLAIN, 24));
   // создаем подходящие наборы атрибутов
   SimpleAttributeSet red = new SimpleAttributeSet();
   StyleConstants.setForeground(red, Color.red);
   StyleConstants.setBold(red, true);
   SimpleAttributeSet blue = new SimpleAttributeSet();
   StyleConstants.setForeground(blue, Color.blue);
   SimpleAttributeSet italic = new SimpleAttributeSet();
   StyleConstants.setItalic(italic, true);
   StyleConstants.setForeground(italic, Color.orange);
   // добавляем текст
   append("Я помню ", null);
   append("чудное", blue);
   append(" мгновенье:\nПередо мной явилась ", null);
   append("ты,", italic);
   append("\nКак ", null);
   append("мимолетное ", red);
   append("виденье", null);
   add(new JScrollPane(textPane), BorderLayout.CENTER);
   setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
protected void append(String s, AttributeSet attributes) {
   Document d = textPane.getDocument();
   try { d.insertString(d.getLength(), s, attributes); }
   catch (BadLocationException ble) {}
   }
   public static void main(String[] args) {
      new Styling().setVisible(true);
   }
}
```

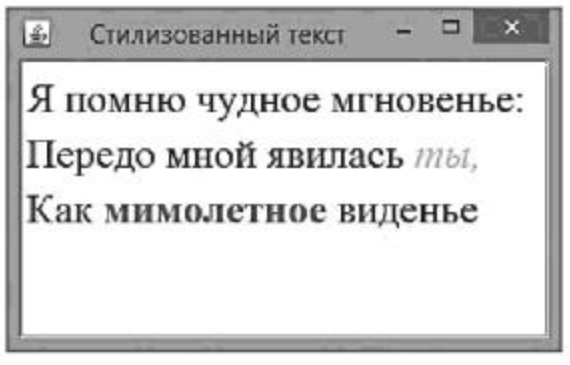

Рис. 18.5. Стилизация текста в компоненте `JTextPane`

Этот код создает объект JTextPane, ссылка на который хранится в поле класса. Здесь также формируется три разных набора атрибутов, которые изменяют стиль и цвет текста. Посредством вспомогательного метода append() текст передается в многострочное поле ввода.

Метод append() добавляет текстовые строки в конец документа, используя предоставленные атрибуты. Помните, если в качестве атрибутов передать значение **null**, компонент JTextPane будет выводить текст с помощью стандартных шрифта и стиля.

При желании вы можете добавить собственный текст. Если поместить текстовый курсор внутрь одного из стилизованных слов, набранный вами текст будет иметь тот же стиль. Неплохо, правда? Вы также могли заметить, что компонент JTextPane имеет встроенную поддержку переноса слов. И так как мы поместили его внутрь контейнера JScrollPane, он будет автоматически выводить полосы прокрутки. Библиотека Swing позволяет без особого труда проделывать по-настоящему интересные трюки. Но подождите — это только начало.

Этот простой пример должен дать вам некоторое представление о возможностях объекта JTextPane. С его помощью можно создавать как простые текстовые процессоры, так и сложные редакторы коммерческого уровня.

Но если вам не хватает возможностей этого компонента или если вы хотите работать с символами, словами и абзацами на более низком уровне, вам никто не мешает самостоятельно заниматься отображением текста, текстового курсора, выделений и т. д. API-интерфейс для рисования двумерной графики предоставляет класс TextLayout, который значительно упрощает данную задачу. Но это уже выходит за рамки нашей книги. Информацию о классе TextLayout, равно как и другие темы, посвященные рисованию текста, можно найти в учебнике «Java 2D Graphics» Джонатана Кнудсена (издательство O'Reilly).

Навигация посредством выделения компонентов

Мы уже много раз затрагивали тему выделения компонентов, упоминая, что обработка и перемещение фокуса в большинстве случаев выполняется автоматически. Это довольно мощный механизм, и на его

работу можно влиять с помощью специальных объектов, которые описывают «политику обхода компонентов» и отвечают за навигацию, выполняемую с помощью клавиатуры. Обычно программист не сталкивается с этим напрямую, но мы все же рассмотрим несколько моментов, о которых вам следует знать.

В библиотеке Swing клавиатурная навигация реализована с помощью класса `KeyboardFocusManager`. Стратегия поиска следующего компонента, который должен быть выделен, определяется объектами `FocusTraversalPolicy`.

В стандартной поставке платформы Java содержится два основных типа этих объектов. Первый, `DefaultFocusTraversalPolicy`, входит в состав пакета AWT. Он эмулирует старый стиль навигации, основанный на порядке добавления компонентов в контейнер. Еще один объект, `LayoutFocusTraversalPolicy`, используется по умолчанию во всех приложениях, написанных с помощью библиотеки Swing. Он анализирует размещение и размер компонентов, пытаясь обеспечить привычную навигацию слева направо и сверху вниз.

Политика обхода компонентов наследуется от родительского контейнера и базируется на «циклах передачи фокуса».

По умолчанию окна и контейнеры типа `JInternalFrame` имеют свои отдельные замкнутые циклы. То есть компоненты внутри них выделяются по кругу (дойдя до последнего элемента управления, фокус возвращается в самое начало) и не могут передать фокус в другое окно (или контейнер `JInternalFrame`).

По умолчанию клавиатурная навигация в библиотеке Swing возможна только в двух направлениях:

Вперед

Tab или **Ctrl+Tab** (второй вариант работает внутри полей ввода)

Назад

Shift+Tab или **Ctrl+Shift+Tab** (второй вариант работает внутри полей ввода)

С помощью метода `setFocusTraversalKeys()`, принадлежащего контейнеру, вы можете назначать собственные сочетания клавиш, которые будут отвечать за навигацию вперед-назад, а также за циклическую пе-

редачу фокуса. Вот пример того, как разрешить прямую навигацию по контейнеру Frame с помощью сочетания клавиш **Ctrl+N**:

```
frame.getFocusTraversalKeys(
   KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS );
AWTKeyStroke ks = AWTKeyStroke.getAWTKeyStroke(
   KeyEvent.VK_N, InputEvent.CTRL_DOWN_MASK );
Set new = new HashSet( old );
set.add( ks );
frame.setFocusTraversalKeys(
   KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS, set);
```

Класс AWTKeyStroke инкапсулирует нажатие клавиши **(N)** и модификатора **(Ctrl)**. Константы, содержащиеся в классе KeyboardFocusManager, определяют направление навигации внутри окна: вперед, назад, вверх и вниз. Напоследок стоит упомянуть, что вы можете перемещать фокус программным способом, используя следующие методы из класса KeyboardFocusManager:

```
focusNextComponent()
focusPreviousComponent()
upFocusCycle()
downFocusCycle()
```

Древовидные списки

Одним из наиболее продвинутых компонентов библиотеки Swing является древовидный список (JTree). Он хорошо подходит для вывода иерархической информации, такой как содержимое жесткого диска или корпоративная структура компании. Модель данных и визуальное представление в древовидном списке, как и в любом другом компоненте, разделены. То есть, вы можете быть уверены, что при изменении содержимого списка компонент обновится соответствующим образом.

JTree — мощный, но сложный объект. Он достаточно большой, чтобы для классов, которые его поддерживают, был выделен отдельный пакет — javax.swing.tree (то же самое касается классов для работы с текстом). Тем не менее древовидный список может быть довольно простым в использовании, если вас устраивает конфигурация, которая предоставляется по умолчанию. На рис. 18.6 показано приложение с компонентом JTree, которое мы рассмотрим позже.

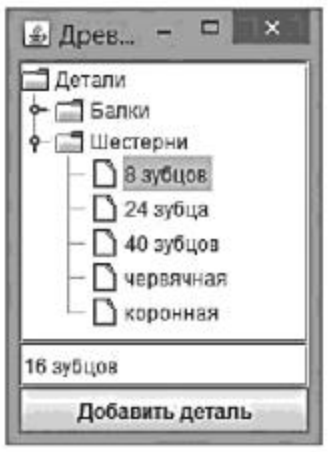

Рис. 18.6. Класс `JTree` в действии

Узлы и модели

Модель данных древовидного списка состоит из взаимосвязанных узлов. Узел имеет имя (обычно, это относится к родительским элементам) и может содержать какое-то количество дочерних узлов. В библиотеке Swing он представлен интерфейсом `TreeNode`. Для узлов, которые можно редактировать, используется интерфейс `MutableTreeNode` (у которого есть стандартная реализация под названием `DefaultMutableTreeNode`). Обычно на вершине иерархии находится только один узел, который называют *корневым*. Модель данных в древовидном списке представлена интерфейсом `TreeModel`, у которого есть стандартная реализация в виде класса `DefaultTreeModel`. Чтобы создать эту модель, достаточно передать в ее конструктор корневой узел типа `TreeNode`. Делается это следующим образом:

```
TreeNode root = new DefaultMutableTreeNode("Корневой узел");
TreeModel model = new DefaultTreeModel(root);
```

Вот еще один реальный пример иерархии. Корневой узел содержит два дочерних узла: «Узел 1» и «Группа». В узле «Группа» находится два вложенных объекта `TreeNode`: «Узел 2» и «Узел 3».

```
MutableTreeNode root = new DefaultMutableTreeNode("Корневой узел");
MutableTreeNode group = new DefaultMutableTreeNode("Группа");
root.insert(group, 0);
root.insert(new DefaultMutableTreeNode("Узел 1"), 1);
group.insert(new DefaultMutableTreeNode("Узел 2"), 0);
group.insert(new DefaultMutableTreeNode("Узел 3"), 1);
```

В качестве второго параметра для метода `insert()` выступает индекс элемента в родительском узле. Организовав свою иерархию, вы можете создать на ее основе модель `TreeModel`, как было показано выше:

```
TreeModel model = new DefaultTreeModel(root);
```

Стандартное поведение древовидного списка

Из готовой модели можно довольно легко получить компонент `JTree`:

```
JTree tree = new JTree(model);
```

Древовидный список ведет себя подобно усовершенствованной версии компонента `JList`. Как видно на рис. 18.6, узлы, которые не содержат других элементов, изображены в виде листов бумаги, тогда как родительские узлы представлены «папками». Если щелкнуть мышью по небольшой «рукоятке» слева от значка, узел раскроется. Того же эффекта можно достичь двойным щелчком по самому узлу. Вы можете выбирать элементы древовидного списка; множественный выбор делается с помощью клавиш **Shift** и **Ctrl**. И, если вы хотите, чтобы компонент `JTree` прокручивался, поместите его внутрь контейнера `JScrollPane` (все как с обычным списком).

События в древовидном списке

Древовидный список генерирует события нескольких видов. Например, с помощью некоторых из них вы можете узнать о раскрытии, закрытии и выделении узлов (когда пользователь щелкает по ним мышью). Для получения этой информации предусмотрено три интерфейса-слушателя:

```
TreeExpansionListener
TreeWillExpandListener
TreeSelectionListener
```

Процесс выбора элементов древовидного списка имеет свои нюансы. Для выбора произвольных узлов при помощи мыши необходимо держать нажатой клавишу **Ctrl**. Класс `TreePath` описывает путь к выбранным элементам, начиная с корневого узла.

В следующем примере показана регистрация объекта слушателя, который выводит на экран последний выбранный узел:

```
tree.addTreeSelectionListener(new TreeSelectionListener() {
   public void valueChanged(TreeSelectionEvent e) {
      TreePath tp = e.getNewLeadSelectionPath();
      System.out.println(tp.getLastPathComponent());
   }
});
```

Готовый пример

Пример, представленный в этом разделе, демонстрирует следующие аспекты работы с древовидными списками:

- Создание древовидной модели на основе класса `DefaultMutableTreeNode`
- Создание и отображение компонента `JTree`
- Отслеживание выбора узлов внутри древовидного списка
- Изменение модели данных с автоматическим обновлением компонента `JTree`

Ниже представлен исходный код этого примера:

```
//файл: PartsTree.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
public class PartsTree {
   public static void main(String[] args) {
      // создаем иерархию узлов
      MutableTreeNode root = new DefaultMutableTreeNode("Детали");
      MutableTreeNode beams = new DefaultMutableTreeNode("Балки");
      MutableTreeNode gears = new DefaultMutableTreeNode("Шестерни");
      root.insert(beams, 0);
      root.insert(gears, 1);
      beams.insert(new DefaultMutableTreeNode("1x4 черная"), 0);
      beams.insert(new DefaultMutableTreeNode("1x6 черная"), 1);
      beams.insert(new DefaultMutableTreeNode("1x8 черная"), 2);
      beams.insert(new DefaultMutableTreeNode("1x12 черная"), 3);
      gears.insert(new DefaultMutableTreeNode("8 зубцов"), 0);
      gears.insert(new DefaultMutableTreeNode("24 зубца"), 1);
      gears.insert(new DefaultMutableTreeNode("40 зубцов"), 2);
      gears.insert(new DefaultMutableTreeNode("червячная"), 3);
      gears.insert(new DefaultMutableTreeNode("коронная"), 4);
      // создаем компонент JTree
      final DefaultTreeModel model = new DefaultTreeModel(root);
      final JTree tree = new JTree(model);
      // создаем поле ввода и кнопку для редактирования модели данных
      final JTextField nameField = new JTextField("16 зубцов");
```

```
final JButton button = new JButton("Добавить деталь");
button.setEnabled(false);
button.addActionListener(new ActionListener(   ) {
   public void actionPerformed(ActionEvent e) {
      TreePath tp = tree.getSelectionPath( );
      MutableTreeNode insertNode =
            (MutableTreeNode)tp.getLastPathComponent(   );
      int insertIndex = 0;
      if (insertNode.getParent(   ) != null) {
         MutableTreeNode parent =
               (MutableTreeNode)insertNode.getParent(   );
         insertIndex = parent.getIndex(insertNode) + 1;
         insertNode = parent;
      }
      MutableTreeNode node =
            new DefaultMutableTreeNode(nameField.getText( ));
      model.insertNodeInto(node, insertNode, insertIndex);
   }
});
JPanel addPanel = new JPanel(new GridLayout(2, 1));
addPanel.add(nameField);
addPanel.add(button);
// отслеживаем выбор узлов
tree.addTreeSelectionListener(new TreeSelectionListener(   ) {
   public void valueChanged(TreeSelectionEvent e) {
      TreePath tp = e.getNewLeadSelectionPath( );
      button.setEnabled(tp != null);
   }
   });
   // создаем контейнер JFrame для хранения древовидного списка
   JFrame frame = new JFrame("Древовидный список");
   frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   frame.setSize(200, 200);
   frame.getContentPane().add(new JScrollPane(tree));
   frame.getContentPane().add(addPanel, BorderLayout.SOUTH);
   frame.setVisible(true);
   }
}
```

В самом начале этого примера мы создаем иерархическую модель данных. Корневой узел называется **Детали**. Он содержит два дочерних

элемента с именами **Балки** и **Шестерни**, в каждом из которых хранится несколько других узлов:

```
MutableTreeNode root = new DefaultMutableTreeNode("Детали");
MutableTreeNode beams = new DefaultMutableTreeNode("Балки");
MutableTreeNode gears = new DefaultMutableTreeNode("Шестерни");
root.insert(beams, 0);
root.insert(gears, 1);
```

Кнопка **Добавить деталь** вставляет элемент в иерархию так, чтобы он находился сразу за текущим узлом (на том же уровне). Имя нового узла можно указать в поле ввода, которое размещается над кнопкой. Чтобы определить, где должен находиться новый элемент, сначала нужно получить текущее выделение. Это делается во внутреннем анонимном классе `ActionListener`:

```
TreePath tp = tree.getSelectionPath();
MutableTreeNode insertNode =
   (MutableTreeNode)tp.getLastPathComponent();
```

Искомая позиция находится в родительском узле текущего элемента (то есть, новый узел будет «братом» того, что выделен в данный момент). Единственная трудность может возникнуть в ситуации, когда текущим является корневой узел, у которого нет «родителя». В таком случае мы определяем порядковый номер выделенного элемента и вставляем новый узел по соседнему индексу:

```
int insertIndex = 0;
if (insertNode.getParent() != null) {
   MutableTreeNode parent =
      (MutableTreeNode)insertNode.getParent();
   insertIndex = parent.getIndex(insertNode) + 1;
   insertNode = parent;
}
MutableTreeNode node =
   new DefaultMutableTreeNode(nameField.getText());
model.insertNodeInto(node, insertNode, insertIndex);
```

Новый узел добавляется в модель данных, а не в сам объект `MutableTableNode`. Для этого используется метод `insertNodeInto()`. Модель сама сообщит компоненту `JTree` о том, что ему пора обновиться.

В этом примере у нас есть еще один обработчик событий, который следит за выделением элементов древовидного списка. Он нужен для

того, чтобы кнопка **Добавить деталь** становилась доступной только тогда, когда у нас появляется выделенный узел:

```
tree.addTreeSelectionListener(new TreeSelectionListener() {
   public void valueChanged(TreeSelectionEvent e) {
      TreePath tp = e.getNewLeadSelectionPath();
      button.setEnabled(tp != null);
   }
});
```

Сразу после запуска этого приложения единственная его кнопка будет неактивной. Доступной для нажатия она станет только после того как вы выберете какой-нибудь элемент; тогда же вы сможете добавлять новые узлы. Чтобы опять сделать кнопку неактивной, отмените выделение текущего элемента, щелкнув по нему мышью с нажатой клавишей **Ctrl**.

Таблицы

Таблицы представляют информацию в виде упорядоченных строк и столбцов. Они хорошо подходят для отображения финансовых отчетов или содержимого реляционных баз данных. Как и древовидные списки, таблицы в библиотеке Swing являются очень мощными и гибкими компонентами, которые довольно легко использовать, благодаря их стандартной конфигурации.

Визуальный компонент, который выводит таблицу на экран, представлен классом `JTable`. Моделью данных для него служит класс `TableModel` из пакета `javax.swing.table` (в этом пакете вы можете найти десятки других вспомогательных классов и интерфейсов).

Раунд первый: сразу в бой

Один из конструкторов класса `JTable` автоматически создает стандартную табличную модель данных на основе массива. Вам нужно всего лишь указать заголовки для столбцов и двумерный массив объектов типа `Object`.

Первый индекс относится к столбцам, а второй к строкам. В следующем примере показано, насколько легко можно создавать таблицы с помощью этого конструктора:

```
//файл: DullShipTable.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
public class DullShipTable {
   public static void main(String[] args) {
      // создаем какие-нибудь табличные данные
      String[] headings =
      new String[] {"Номер", "Срочно?", "Откуда",
               "Куда", "Дата отправки", "Вес" };
      Object[][] data = new Object[][] {
{ "100420", Boolean.FALSE, "Свердловск", "Москва",
   "02/06/2013", new Float(450) },
{ "202174", Boolean.TRUE, "Воронеж", "Астрахань",
   "05/20/2013", new Float(1250) },
{ "450877", Boolean.TRUE, "Волгоград", "Самара",
   "03/20/2013", new Float(1745) },
{ "101891", Boolean.FALSE, "Казань", "Пермь",
   "04/04/2013", new Float(88) }
};
// создаем модель данных и компонент JTable
JTable table = new JTable(data, headings);
JFrame frame = new JFrame("Таблица перевозок v1.0");
frame.add(new JScrollPane(table));
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setSize(500, 200);
frame.setVisible(true);
   }
}
```

Это небольшое приложение выводит таблицу, показанную на рис. 18.7.

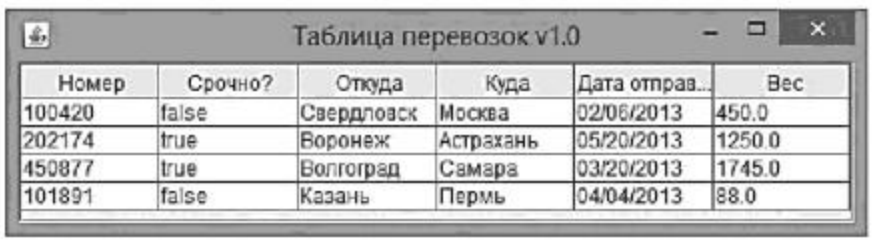

Рис. 18.7. Простейшая таблица

Мы получили довольно впечатляющий результат, несмотря на очень компактный код. Это стало возможным благодаря тому, что многие операции библиотека Swing выполняет автоматически. Например:

Заголовки для столбцов

Компонент `JTable` автоматически выводит заголовки, которые визуально отличаются от обычных столбцов. Пользователь сразу может понять, что они не являются частью той области, в которой выводятся данные.

Переполнение ячеек

Текст, который не помещается в ячейку, автоматически урезается, а в конце него выводится многоточие (...). Это можно наблюдать на примере заголовка «Дата отправки», который показан на рис. 18.7.

Выбор строк

Чтобы выбрать целую строку, достаточно щелкнуть мышью по одной из ее ячеек. Но это поведение можно изменить с помощью методов `setCellSelectionEnabled()`, `setColumnSelectionAllowed()` и `setRowSelectionAllowed()`, которые позволяют выбирать отдельные ячейки, строки, столбцы или какое-то их сочетание.

Редактирование ячеек

Если вы сделаете двойной щелчок мышью по ячейке, у вас появится возможность редактировать ее содержимое. Вы увидите поле ввода с мигающим текстовым курсором, куда можно будет вводить новые данные.

Изменение размеров столбцов

Указатель мыши, размещенный между двумя заголовками, приобретает вид горизонтальной двунаправленной стрелки.

Если нажать при этом левую клавишу мыши и передвинуть указатель, можно изменить размер столбца, который находится слева; ширина других столбцов тоже может поменяться, но это зависит от того, как сконфигурирован компонент `JTable`. Этим процессом можно управлять с помощью метода `setAutoResizeMode()`.

Перемещение столбцов

Вы можете переместить в другую часть таблицы целый столбец, если щелкните мышью по его заголовку и отведете указатель в сторону.

Можете сами попрактиковаться. Это весело!

Раунд второй: создание модели данных

`JTable` является очень мощным компонентом. Он автоматически предоставляет множество разных удобств. Тем не менее его стандартная конфигурация не совсем подходит для нашего простого примера. В частности, нам бы не хотелось, чтобы пользователь мог изменять содержимое ячеек. Также было бы неплохо выводить в столбце **Срочно?** флажки, а не обычный текст. Кроме того, можно было бы отформатировать подходящим образом числа, содержащиеся в столбце **Вес**.

Чтобы сделать компонент `JTable` более гибким, мы напишем для него свою собственную модель данных, основанную на интерфейсе `TableModel`. К счастью, библиотека Swing содержит класс под названием `AbstractTableModel`, который значительно упрощает эту задачу. Мы просто наследуем этот класс и переопределим те его аспекты, которые нам бы хотелось изменить.

Все производные класса `AbstractTableModel` должны реализовывать как минимум три следующих метода:

```
public int getRowCount(),public int getColumnCount()
```

Возвращают количество строк и столбцов в текущей модели данных.

```
public Object getValueAt(int row , int column )
```

Возвращает значение заданной ячейки.

Когда таблице нужно получить какое-то определенное значение, она вызывает из модели данных метод `getValueAt()`. Для определения общего размера таблицы используются методы `getRowCount()` и `getColumnCount()`.

Простейшая табличная модель выглядит следующим образом:

```
public static class ShipTableModel extends AbstractTableModel {
   private Object[][] data = new Object[][] {
      { "100420", Boolean.FALSE, "Свердловск", "Москва",
         "02/06/2013", new Float(450) },
      { "202174", Boolean.TRUE, "Воронеж", "Астрахань",
         "05/20/2013", new Float(1250) },
      { "450877", Boolean.TRUE, "Волгоград", "Самара",
         "03/20/2013", new Float(1745) },
```

```
      { "101891", Boolean.FALSE, "Казань", "Пермь",
        "04/04/2013", new Float(88) }
   };
   public int getRowCount() { return data.length; }
   public int getColumnCount() { return data[0].length; }
   public Object getValueAt(int row, int column) {
      return data[row][column];
   }
}
```

Было бы неплохо использовать в нашей модели данных заголовки столбцов из предыдущего примера. Этого можно добиться с помощью метода `getColumnName()` из интерфейса `TableModel`:

```
private String[] headings = new String[] {
   "Номер", "Срочно?", "Откуда", "Куда", "Дата отправки", "Вес"
};
public String getColumnName(int column) {
   return headings[column];
}
```

По умолчанию модель `AbstractTableModel` запрещает редактирование всех своих ячеек — то, чего мы хотели с самого начала. Для этого не нужно писать никакого дополнительного кода.

Напоследок сделаем так, чтобы столбцы **Срочно?** и **Вес** форматировали свои значения специальным образом. Для этого наша табличная модель должна иметь кое-какое представление о типах хранящихся в ней данных.

Компонент `JTable` автоматически генерирует ячейки с флажками для столбцов типа `Boolean`, а также форматирует значения в столбцах числовых типов. Чтобы табличная модель могла определять типы данных в каждом из своих столбцов, мы должны переопределить метод `getColumnClass()`, который вызывается именно для этих целей и способен представлять информацию специальным образом. Мы будем возвращать класс элемента из первой строки:

```
public Class getColumnClass(int column) {
   return data[0][column].getClass();
}
```

Вот и все, что нужно было сделать. Ниже представлена окончательная версия этого примера, в которой показано, как с помощью вы-

шеописанных приемов создать таблицу на основе собственной модели данных:

```
//файл: ShipTable.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
public class ShipTable {
   public static class ShipTableModel extends AbstractTableModel {
      private String[] headings = new String[] {
         "Номер", "Срочно?", "Откуда", "Куда", "Дата отправки", "Вес"
      };
      private Object[][] data = new Object[][] {
         { "100420", Boolean.FALSE, "Свердловск", "Москва",
           "02/06/2013", new Float(450) },
         { "202174", Boolean.TRUE, "Воронеж", "Астрахань",
           "05/20/2013", new Float(1250) },
         { "450877", Boolean.TRUE, "Волгоград", "Самара",
           "03/20/2013", new Float(1745) },
         { "101891", Boolean.FALSE, "Казань", "Пермь",
           "04/04/2013", new Float(88) }
      };
      public int getRowCount() { return data.length; }
      public int getColumnCount() { return data[0].length; }
      public Object getValueAt(int row, int column) {
         return data[row][column];
      }
      public String getColumnName(int column) {
         return headings[column];
      }
      public Class getColumnClass(int column) {
         return data[0][column].getClass();
      }
   }
   public static void main(String[] args)
   {
      // создаем модель данных и компонент JTable
      TableModel model = new ShipTableModel();
      JTable table = new JTable(model);
      table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
```

```
        JFrame frame = new JFrame("Таблица перевозок v1.0");
        frame.getContentPane().add(new JScrollPane(table));
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize(500, 200);
        frame.setVisible(true);
    }
}
```

После запуска это приложение будет выглядеть, как на рис. 18.8.



Рис. 18.8. Настроенная таблица

Раунд третий: простая электронная таблица

Чтобы продемонстрировать, насколько действенным и гибким может быть результат разделения данных и пользовательского интерфейса, мы создадим более сложную модель. В следующем примере мы реализуем очень компактную, но мощную электронную таблицу (см. рис. 18.9), не внося почти никаких изменений в компонент `JTable`. Вся обработка данных будет происходить внутри модели данных под названием `SpreadSheetModel`.



Рис. 18.9. Простая электронная таблица

Наше приложение умеет делать обычные для любой электронной таблицы вещи — например, оно позволяет вводить в ячейки числа или математические выражения, такие как `(A1*B2)+C3`.* Редактирование и обновление ячеек происходит в стандартном компоненте `JTable`. Мы реализовали методы, необходимые для присваивания и извлечения дан-

* Чтобы начать редактирование ячейки, дважды щелкните по ней мышью.

ных. Однако без необходимых проверок нашу таблицу легко сломать (например, мы не выполняем проверку на циклические зависимости, наличие которых очень нежелательно).

Как вы увидите дальше, большая часть кода этого приложения приходится на внутренний класс, который используется для анализа и вычисления уравнений в ячейках. Если вас не интересует этот аспект программирования, можете пролистывать дальше. Но если вы еще никогда не сталкивались с таким подходом к разбору математических выражений, вам этот код может показаться довольно любопытным. Используя всю магию рекурсии и мощь класса `String`, мы реализовали систему разбора основных арифметических операций с произвольным количеством вложенных скобок, и при этом уложились всего в 50 строчек кода.

А вот и сам код:

```
//файл: SpreadsheetModel.java
import java.util.StringTokenizer;
import javax.swing.*;
import javax.swing.table.AbstractTableModel;
public class SpreadsheetModel extends AbstractTableModel {
   Expression [][] data;
   public SpreadsheetModel( int rows, int cols ) {
      data = new Expression [rows][cols];
   }
   public void setValueAt(Object value, int row, int col) {
      data[row][col] = new Expression( (String)value );
      fireTableDataChanged( );
   }
   public Object getValueAt( int row, int col ) {
      if ( data[row][col] != null )
         try { return data[row][col].eval( ) + ""; }
         catch ( BadExpression e ) { return "Ошибка"; }
      return "";
   }
   public int getRowCount( ) { return data.length; }
   public int getColumnCount( ) { return data[0].length; }
   public boolean isCellEditable(int row, int col) { return true; }
   class Expression {
      String text;
      StringTokenizer tokens;
      String token;
```

```
Expression( String text ) { this.text = text.trim(   ); }
float eval( ) throws BadExpression {
   tokens = new StringTokenizer( text, " */+-()", true );
   try { return sum( ); }
   catch ( Exception e ) { throw new BadExpression( ); }
}
private float sum( ) {
   float value = term(   );
   while( more( ) && match("+-") )
      if ( match("+") ) { consume(); value = value + term( ); }
      else { consume(); value = value - term( ); }
   return value;
}
private float term(   ) {
   float value = element(   );
   while( more( ) && match( "*/") )
      if ( match("*") ) { consume(); value = value * element( ); }
      else { consume(); value = value / element( ); }
   return value;
}
private float element(   ) {
   float value;
   if ( match( "(") ) { consume(); value = sum(   ); }
   else {
      String svalue;
      if ( Character.isLetter( token().charAt(0) ) ) {
      int col = findColumn( token(   ).charAt(0) + "" );
      int row = Character.digit( token( ).charAt(1), 10 );
      svalue = (String)getValueAt( row, col );
   } else
      svalue = token( );
      value = Float.parseFloat( svalue );
   }
   consume( ); // ")" или составная часть выражения
   return value;
}
private String token( ) {
   if ( token == null )
      while ( (token=tokens.nextToken(  )).equals(" ") );
   return token;
}
```

```
        private void consume( ) { token = null; }
        private boolean match( String s ) { return s.indexOf( token(   )
)!=-1; }
        private boolean more() { return tokens.hasMoreTokens(   ); }
    }
    class BadExpression extends Exception { }
    public static void main( String [] args ) {
        JFrame frame = new JFrame("Эксельсиор!");
        JTable table = new JTable( new SpreadsheetModel(15, 5) );
        table.setPreferredScrollableViewportSize( table.getPreferredSize()
);
        table.setCellSelectionEnabled(true);
        frame.getContentPane().add( new JScrollPane( table ) );
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.pack();
    frame.setVisible(true);
    }
}
```

Наша модель наследует класс `AbstractTableModel` и переопределяет всего несколько его методов. Данные хранятся в двумерном массиве в виде объектов `Expression`, которые с помощью метода `setValueAt()` создаются на основе строк, введенных пользователем. Этот метод возвращает значение ячейки, предварительно делая вызов `eval()` из хранящегося в ней выражения. Если пользователь введет некорректные данные, будет сгенерировано исключение `BadExpression`, а в ячейке появится слово «Ошибка». Чтобы определить размерность электронной таблицы и позволить пользователю редактировать ее поля, нам необходимо переопределить три метода из интерфейса `TableModel`: `getRowCount()`, `getColumnCount()` и `isCellEditable()`. Вот и все! Вспомогательный метод `findColumn()` наследуется из класса `AbstractTableModel`.

Теперь перейдем к самому интересному. С помощью уже известного нам класса `StringTokenizer` мы последовательно разбиваем строку с выражением на отдельные значения и математические символы (`+-*/()`). Затем полученные элементы обрабатываются тремя методами, `sum()`, `term()` и `element()`, которые вызываются по цепочке, один из другого. Давайте рассмотрим их в обратном порядке, начиная с последнего.

На самом нижнем уровне находится метод `element()`, который считывает отдельные числовые значения или имена ячеек (например, `5.0` или `B2`). Метод `term()`, работающий уровнем выше, обрабатывает зна-

чения, возвращаемые методом `element()`; он отвечает за все операции умножения и деления. Затем результат передается в самое начало цепочки вызовов, в метод `sum()`, где выполняются сложение и вычитание. Если метод `element()` встречает скобки, он делает вызов `sum()`, чтобы обработать вложенное выражение. В итоге вместо скобок мы получаем результат выполнения вложенных операций (иногда после нескольких рекурсивных проходов), который и возвращается методом `element()`. Таким образом, вложенные выражения распутываются за счет рекурсии. Еще один неочевидный момент касается порядка следования трех вышеперечисленных методов. Здесь действует приоритет арифметических операторов; то есть, первыми (на самом нижнем уровне) анализируются «атомарные» значения, за ними следуют операции умножения/деления, и в самом конце выполняются сложение и вычитание.

Грамматический анализ основан на четырех простых методах, которые делают код более простым для восприятия: `token()`, `consume()`, `match()` и `more()`. Первый из них использует объект `StringTokenizer`, чтобы получить следующее значение, которое потом сравнивается с указанным текстом при помощи метода `match()`. Метод `consume()` используется для перехода к следующей составной части выражения, а метод `more()` сообщает о том, что все значения уже проанализированы.

Сортировка и фильтрация

С выходом платформы Java 6 у компонента `JTable` появились новые, простые в использовании механизмы сортировки и фильтрации. В следующем приложении демонстрируется применение стандартного класса `TableRowSorter`, а также пример простого фильтра на основе регулярных выражений.

```
//файл: SortFilterTable.java
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
import java.awt.BorderLayout;
import java.util.regex.PatternSyntaxException;
public class SortFilterTable extends JFrame {
   private JTable table;
   private JTextField filterField;
   public SortFilterTable() {
      super("Сортировка и фильтрация таблицы");
```

```
setSize(500, 200);
setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
// Создаем простую модель данных для таблицы
TableModel model = new AbstractTableModel() {
   private String[] columns = {"Имя",
      "Домашнее животное", "Количество детей"};
   private Object[][] people = {
               {"Иван Сидоров", "хамелеон", 1},
               {"Борис Иванов", "сахарная сумчатая летяга", 2},
               {"Валентин Петров", "собака", 3},
               {"Валентина Петрова", "панда", 2}
               };
   public int getColumnCount() { return columns.length; }
   public int getRowCount() { return people.length; }
   public Object getValueAt(int row, int col) {
      return people[row][col];
   }
   public Class getColumnClass(int col) {
      return getValueAt(0, col).getClass();
   }
};
table = new JTable(model);
table.setAutoCreateRowSorter(true);
table.setFillsViewportHeight(true);
// Создаем область для ввода фильтров
JPanel filterPanel = new JPanel(new BorderLayout());
JLabel filterLabel = new JLabel("Фильтр ",
   SwingConstants.TRAILING);
filterPanel.add(filterLabel, BorderLayout.WEST);
filterField = new JTextField();
filterLabel.setLabelFor(filterField);
filterPanel.add(filterField);
// Применяем фильтр при изменении содержимого
// поля ввода filterField
filterField.getDocument().addDocumentListener(
   new DocumentListener() {
      public void changedUpdate(DocumentEvent e) {
         filter();
      }
      public void insertUpdate(DocumentEvent e) {
         filter();
```

```
            }
            public void removeUpdate(DocumentEvent e) {
               filter();
            }
      });
      filterPanel.setBorder(BorderFactory.createEmptyBorder(2,
         2, 2, 2));
      add(filterPanel, BorderLayout.NORTH);
      add(new JScrollPane(table));
   }
   // Фильтруем по первому столбцу
   private void filter() {
      RowFilter<TableModel, Object> filter = null;

      // Обновляем фильтр, если пользователь ввел
      // корректное регулярное выражение
      try {
         // Применяем фильтр к столбцам с индексами 0 и 1
         filter = RowFilter.regexFilter(filterField.getText(),
            0, 1);
      } catch (PatternSyntaxException e) {
         return;
      }
      ((TableRowSorter)table.getRowSorter()).setRowFilter(
         filter);
   }
   public static void main(String[] args) {
      new SortFilterTable().setVisible(true);
   }
}
```

Попробуйте отсортировать один из столбцов, щелкнув мышью по его заголовку. Мы используем стандартный механизм сортировки, согласно которому элементы размещаются естественным образом (строки — в алфавитном порядке, числа — от меньшего к большему). Вы можете переопределить принцип сортировки; для этого необходимо реализовать свой собственный компаратор и передать его объекту `TableRowSorter`:

```
TableRowSorter<TableModel> reverseSorter
   = new TableRowSorter<TableModel>(table.getModel());
reverseSorter.setComparator(0, new Comparator<String>() {
```

```
    public int compare(String a, String b) {
        return -a.compareTo(b);
    }
});
table.setRowSorter(reverseSorter);
```

Если вам нужно выполнять более сложную сортировку, вы можете наследовать класс `TableRowSorter` (или его родителя, класс `DefaultRowSorter`).

При наборе текста в поле ввода, размещенного над таблицей, к значениям ячеек из первых двух столбцов (с индексами 0 и 1) будет применяться фильтр, в котором введенные данные используются в качестве регулярных выражений. Например, если вы введете слово «Петров», в таблице останется только две записи — «Валентин Петров» и «Валентина Петрова».

Печать таблиц

Библиотека Swing максимально упрощает распечатку содержимого таблиц. Думаете, мы приукрашиваем? Если вас устраивает стандартная конфигурация, то вызов диалогового окна для печати будет выглядеть следующим образом:

```
myJTable.print();
```

Вот и все. По умолчанию таблица растягивается до ширины печатной страницы. Этот режим называется «по ширине».

Данным процессом можно управлять с помощью перечисления `PrintMode` из класса `JTable`, а именно, посредством значений `NORMAL` и `FIT_WIDTH`:

```
table.print( JTable.PrintMode.NORMAL );
```

Режим `NORMAL` (забавно, что он не используется по умолчанию) позволяет печатать таблицу на нескольких страницах, разбивая ее по горизонтали без уменьшения размеров. В обоих вариантах строки могут занимать несколько страниц по вертикали.

Есть и другие версии метода `print()`, которые позволяют лучше контролировать процесс печати, а также выводить вверху и внизу страницы собственные колонтитулы. Немного больше о печати данных мы поговорим в главе 20, когда будем обсуждать двумерную графику.

Рабочие столы

Сейчас вы, наверное, думаете, что на этом возможности библиотеки Swing уж точно исчерпываются. Но самое интересное еще впереди. Если вы когда-нибудь мечтали организовать свою небольшую оконную систему, в этом вам помогут компоненты `JDesktopPane` и `JInternalFrame`. То, как это выглядит, показано на рис. 18.10.

Большую часть функций выполняет класс `JInternalFrame`. Он представляет внутренние окна, которые можно перемещать (если потянуть за заголовок), сворачивать в небольшой значок, разворачивать на весь рабочий стол (`JDesktopPane`, а не системный) и масштабировать (если потянуть за обрамление). С точки зрения программиста, внутреннее окно является специальным видом контейнера, внутри которого можно размещать визуальные компоненты приложения.

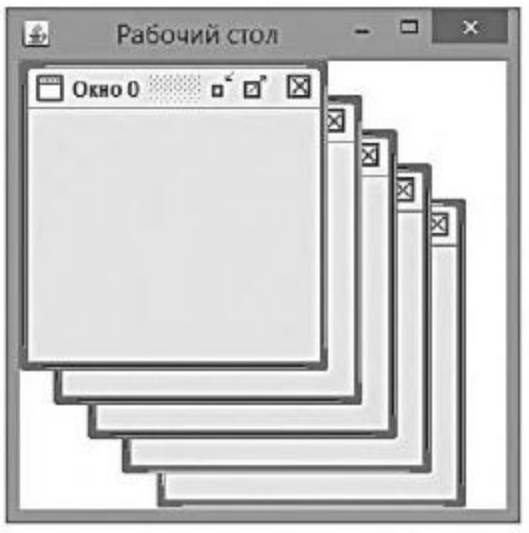

Рис. 18.10. Использование внутренних окон в рамках компонента `JDesktopPane`

С помощью следующего кода можно создать окна, показанные на рис. 18.10:

```
//файл: Desktop.java
import javax.swing.*;
public class Desktop {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Рабочий стол");
      JDesktopPane desktop = new JDesktopPane();
      for (int i = 0; i < 5; i++) {
         JInternalFrame internal =
            new JInternalFrame("Окно " + i, true, true, true, true);
         internal.setSize(180, 180);
         internal.setLocation(i * 20, i * 20);
         internal.setVisible(true);
```

```
            desktop.add(internal);
        }
        frame.setSize(300, 300);
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setContentPane(desktop);
        frame.setVisible(true);
    }
}
```

Все, что мы здесь сделали — это создали объект `JDesktopPane` и добавили в него несколько внутренних окон. В конструктор каждого контейнера `JInternalFrame` передается заголовок и четыре значения `true`, которые сигнализируют о том, что окно можно масштабировать, закрывать, сворачивать и разворачивать.

Объекты `JInternalFrame` генерируют свой собственный набор событий. Однако стоит сказать, что типы `InternalFrameEvent` и `InternalFrameListener` отличаются от своих аналогов, `WindowEvent` и `WindowListener`, исключительно названиями. Если вы хотите отслеживать закрытие внутреннего окна, вам нужно всего лишь зарегистрировать объект-слушатель `InternalFrameListener` и реализовать для него метод `internalFrameClosing()`. То же самое мы делали для обычного окна `JFrame`, когда реализовывали метод `windowClosing()`.

Изменение внешнего вида

Мы уже упоминали, что компоненты библиотеки Swing могут легко и непринужденно изменять свое оформление. В целом, все элементы управления в рамках одной программы, так или иначе, имеют общие черты. Например, все они, скорее всего, используют один шрифт и общую цветовую гамму. Внешний вид и восприятие элементов пользовательского интерфейса определяются набором визуальных стилей.

Любая современная операционная система, предназначенная для настольных компьютеров, имеет свое уникальное оформление. Например, OS X выглядит не так, как Windows. Платформа Java предлагает несколько готовых визуальных стилей для компонентов библиотеки Swing. Но если вы увлекаетесь графическим дизайном, никто не мешает вам создать и применить к своему приложению собственный стиль. Механизм, который позволяет с легкостью менять внешний вид компонентов, реализован в виде пакета `javax.swing.plaf`.

Лучше один раз увидеть, чем сто раз услышать. В следующем примере мы создадим несколько компонентов, оформление которых можно будет изменять динамически с помощью команд меню:

```
//файл: QuickChange.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class QuickChange extends JFrame {
   public QuickChange() {
      super("Изменение внешнего вида компонентов");
      createGUI();
   }
protected void createGUI(   ) {
   setSize(300, 200);
   // Создаем простое меню "Файл"
   JMenu file = new JMenu("Файл", true);
   JMenuItem quit = new JMenuItem("Выход");
   file.add(quit);
   quit.addActionListener(new ActionListener( ) {
      public void actionPerformed(ActionEvent e) { System.exit(0); }
   });
   // Создаем меню "Внешний вид"
   JMenu lnf = new JMenu("Внешний вид", true);
   ButtonGroup buttonGroup = new ButtonGroup( );
   final UIManager.LookAndFeelInfo[] info =
         UIManager.getInstalledLookAndFeels( );
   for (int i = 0; i < info.length; i++) {
      JRadioButtonMenuItem item = new
         JRadioButtonMenuItem(info[i].getName(), i == 0);
      final String className = info[i].getClassName( );
      item.addActionListener(new ActionListener( ) {
         public void actionPerformed(ActionEvent ae) {
            try { UIManager.setLookAndFeel(className); }
            catch (Exception e) { System.out.println(e); }
            SwingUtilities.updateComponentTreeUI(QuickChange.this);
         }
      });
      buttonGroup.add(item);
      lnf.add(item);
   }
```

```
// Добавляем строку меню
JMenuBar mb = new JMenuBar( );
mb.add(file);
mb.add(lnf);
setJMenuBar(mb);
// Добавляем несколько компонентов
JPanel jp = new JPanel(  );
jp.add(new JCheckBox("JCheckBox"));
String[] names =
   new String[] { "Тоска", "Каварадосси", "Скарпиа",
         "Анджелотти", "Сполетта", "Шарроне",
         "Тюремщик", "Ризничий", "Пастух" };
   jp.add(new JComboBox(names));
   jp.add(new JButton("JButton"));
   jp.add(new JLabel("JLabel"));
   jp.add(new JTextField("JTextField"));
   JPanel main = new JPanel(new GridLayout(1, 2));
   main.add(jp);
   main.add(new JScrollPane(new JList(names)));
   setContentPane(main);
setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
public static void main(String[] args) {
   new QuickChange().setVisible(true);
}
}
```

Наиболее интересным аспектом этого приложения является процесс создания меню. Сначала с помощью класса `UIManager` мы получаем список визуальных стилей, доступных на текущем компьютере:

```
final UIManager.LookAndFeelInfo[] info =
   UIManager.getInstalledLookAndFeels();
```

Информация о визуальных стилях возвращается в виде объектов типа `UIManager.LookAndFeelInfo`. Несмотря на такое длинное название, они не представляют собой ничего интересного; их главная задача — связать название стиля (например, «Metal») с именем класса, который его реализует (например, `javax.swing.plaf.metal.MetalLookAndFeel`). В вышеприведенном примере мы используем эти названия для создания пунктов меню. Когда пользователь выбирает один из пунктов, мы обращаемся к классу `UIManager`, чтобы применить соответствующий стиль.

После этого мы вызываем статический метод `updateComponentTreeUI()` из класса `SwingUtilities`, чтобы обновить все компоненты.

Пакет JDK включает в себя несколько визуальных стилей: Windows, OS X, Motif, Metal (оригинальный), Synth (очень гибкий в настройке) и самый новый, Nimbus. Первые три стиля эмулируют соответствующие графические окружения. Например, в операционной системе OS X по умолчанию выбирается оформление пользовательского интерфейса под названием Aqua, которое используется во всех современных приложениях для компьютеров Macintosh. К сожалению, из-за лицензионных ограничений этот стиль недоступен на других платформах (то же самое касается оформления операционных систем семейства Windows).

Стиль Metal (в частности, его тема Ocean) является стандартным для некоторых платформ, но на фоне современных оконных систем он выглядит довольно невзрачно. Есть куда более приемлемый вариант под названием Nimbus, который по своему внешнему виду ничуть не уступает новейшим графическим системам. Кроме того, в нем используется масштабируемая векторная графика, благодаря чему он легко адаптируется к экранам с разными размерами и высокой плотностью точек.

Стиль Synth, который является прародителем Nimbus, вобрал в себя некоторые новые тенденции по созданию пользовательских интерфейсов с изменяемым визуальным оформлением. Многие современные приложения позволяют легко изменять свой внешний вид и использовать для создания новых тем оформления обычные изображения и простые файлы с настройками. Такой подход не может сравниться с полноценным механизмом визуальных стилей, но он предоставляет множество возможностей, которые не требуют изменять код программы. Synth мало чем отличается от любого другого визуального стиля, однако он позволяет влиять на внешний вид приложения при помощи изображений и XML-файлов. Например, чтобы обрамление компонентов выглядело блестящим и «металлическим» (как в стиле Metal), вы можете предоставить соответствующие графические файлы, указав смещение для внутренних «углов» и способ заливки больших поверхностей (растягивание или многократное повторение). В результате, стиль Synth будет использовать это обрамление для любых указанных вами компонентов. Благодаря такому подходу можно делать много разных вещей; если ваш пользовательский интерфейс станет слишком сложным, вы сможете опять вернуться к написанию кода и указать объекты, которые будут отвечать за отрисовку ваших компонентов.

Создание нестандартных компонентов

И в этой, и в предыдущей главе были рассмотрены разные элементы пользовательского интерфейса. Мы использовали богатый набор инструментов библиотеки Swing, складывали вместе отдельные компоненты, словно кирпичики, расширяли их возможности, но так и не создали ничего нового. В этом разделе мы напишем с нуля свой собственный компонент — *круговой регулятор*.

До этого момент все наши приложения были достаточно автономными; они сами знали, что им делать, и не использовали в своей работе дополнительные модули. В примере с меню класс `DinnerFrame` содержал не только сами пункты, но и код, который реагирует на их выбор. Если бы мы захотели как-то изменить логику обработки пунктов меню, нам бы пришлось редактировать наш класс. Настоящие компоненты разделяют отслеживание действий пользователя от их обработки. Это позволяет не только реагировать на события, но и оповещать о них другие части приложения.

Генерирование событий

Чтобы наши классы могли считаться полноценными компонентами, они должны взаимодействовать между собой стандартным для компонентов способом: посредством генерирования событий и передачи их объектам-слушателям. До этого момента мы видели множество примеров кода, который занимается обработкой событий; теперь пришло время заняться их генерированием.

Может это и звучит сложно, но на самом деле ничего особенного в этом нет. Вы можете либо создавать новые виды событий на основе класса `java.util.EventObject`, либо использовать уже имеющиеся типы. В любом случае вам необходимо разрешить регистрацию объектов-слушателей и предоставить средства для доставки своих событий по назначению. Класс `JComponent` предоставляет защищенное поле `listenerList`, с помощью которого вы можете вести список зарегистрированных реализаций интерфейса `EventListener`. Это экземпляр класса `EventListenerList`, который следит за обработчиками событий, отсортированных по их типу; фактически, он ведет себя словно метрдотель в ресторане.

Обычно в создании новых типов событий нет никакой необходимости. Класс `JComponent` содержит методы, которые поддерживают ге-

нерирование универсальных объектов `PropertyChangeEvent` во время изменения свойств компонента. Ниже мы рассмотрим пример, в котором события этого типа передаются при редактировании текущего значения.

Круговой регулятор

В стандартной поставке библиотеки Swing нет ничего похожего на старомодный круговой регулятор (такие еще использовались в радиоприемниках для регулировки громкости). Эту нишу в современных интерфейсах занял компонент `JSlider`. В данном разделе мы создадим класс `Dial`; чтобы изменить его текущее значение, нужно щелкнуть по нему мышью и перетащить указатель «круговым движением» (см. рис. 18.11). При этом будут генерироваться события типа `DialEvent`. Круговой регулятор можно использовать так же, как любой другой компонент библиотеки Swing. У него даже будет собственный интерфейс для прослушивания событий, `DialListener`.

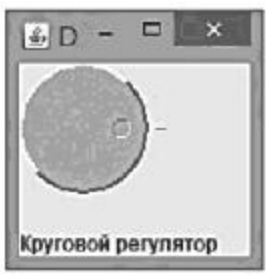

Рис. 18.11. Компонент `Dial`

Ниже представлен код класса `Dial`:

```
//файл: Dial.java
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
public class Dial extends JComponent {
   int minValue, nvalue, maxValue, radius;
   public Dial() { this(0, 100, 0); }
   public Dial(int minValue, int maxValue, int value) {
      setMinimum( minValue );
      setMaximum( maxValue );
      setValue( value );
      setForeground( Color.lightGray );
      addMouseListener(new MouseAdapter() {
```

```
        public void mousePressed(MouseEvent e) { spin(e); }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) { spin(e); }
    });
}
protected void spin( MouseEvent e ) {
int y = e.getY();
int x = e.getX();
double th = Math.atan((1.0 * y - radius) / (x - radius));
int value=(int)(th / (2 * Math.PI) * (maxValue - minValue));
if (x < radius)
    setValue( value + (maxValue-minValue) / 2 + minValue);
else if (y < radius)
    setValue( value + maxValue );
else
    setValue( value + minValue);
}
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    int tick = 10;
    radius = Math.min( getSize().width,getSize().height )/2 - tick;
    g2.setPaint( getForeground().darker() );
    g2.drawLine( radius * 2 + tick / 2, radius,
        radius * 2 + tick, radius);
    g2.setStroke( new BasicStroke(2) );
    draw3DCircle( g2, 0, 0, radius, true );
    int knobRadius = radius / 7;
    double th = nvalue * (2 * Math.PI) / (maxValue - minValue);
    int x = (int)(Math.cos(th) * (radius - knobRadius * 3)),
    y = (int)(Math.sin(th) * (radius - knobRadius * 3));
    g2.setStroke(new BasicStroke(1));
    draw3DCircle(g2, x + radius - knobRadius,
        y + radius - knobRadius, knobRadius, false );
}
private void draw3DCircle(
    Graphics g, int x, int y, int radius, boolean raised)
{
    Color foreground = getForeground();
    Color light = foreground.brighter();
    Color dark = foreground.darker();
```

```
        g.setColor(foreground);
        g.fillOval(x, y, radius * 2, radius * 2);
        g.setColor(raised ? light : dark);
        g.drawArc(x, y, radius * 2, radius * 2, 45, 180);
        g.setColor(raised ? dark : light);
        g.drawArc(x, y, radius * 2, radius * 2, 225, 180);
    }
    public Dimension getPreferredSize() {
        return new Dimension(100, 100);
    }
    public void setValue( int value ) {
        this.nvalue = value - minValue;
        repaint();
        fireEvent();
    }
    public int getValue() { return nvalue+minValue; }
    public void setMinimum(int minValue) { this.minValue = minValue; }
    public int getMinimum() { return minValue; }
    public void setMaximum(int maxValue) { this.maxValue = maxValue; }
    public int getMaximum() { return maxValue; }
    public void addDialListener(DialListener listener) {
        listenerList.add( DialListener.class, listener );
    }
    public void removeDialListener(DialListener listener) {
        listenerList.remove( DialListener.class, listener );
    }
    void fireEvent() {
        Object[] listeners = listenerList.getListenerList();
        for ( int i = 0; i < listeners.length; i += 2 )
            if ( listeners[i] == DialListener.class )
                ((DialListener)listeners[i + 1]).dialAdjusted(
                    new DialEvent(this, getValue()) );
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame("Dial v1.0");
        final JLabel statusLabel = new JLabel("Круговой регулятор");
        final Dial dial = new Dial();
        frame.add(dial, BorderLayout.CENTER);
        frame.add(statusLabel, BorderLayout.SOUTH);
        dial.addDialListener(new DialListener() {
            public void dialAdjusted(DialEvent e) {
```

```
            statusLabel.setText("Значение: " + e.getValue());
         }
      });
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize( 150, 150 );
      frame.setVisible( true );
   }
}
```

Код события `DialEvent`, которое является производной класса `java.util.EventObject`:

```
//файл: DialEvent.java
import java.awt.*;
public class DialEvent extends java.util.EventObject {
   int value;
   DialEvent( Dial source, int value ) {
      super( source );
      this.value = value;
   }
   public int getValue() {
      return value;
   }
}
```

И, наконец, код интерфейса `DialListener`:

```
//файл: DialListener.java
public interface DialListener extends java.util.EventListener {
   void dialAdjusted( DialEvent e );
}
```

Давайте начнем с класса `Dial`. Мы сосредоточим свое внимание на основных структурных моментах, а всеми тонкостями тригонометрии вы можете заняться самостоятельно.

В методе `main()` демонстрируется принцип использования кругового регулятора для построения пользовательского интерфейса. В нем мы создаем объект `Dial` и помещаем его в контейнер `JFrame`.

Затем мы регистрируем объект-слушатель для нашего компонента. При получении события `DialEvent` мы извлекаем из объекта `Dial` текущее значение и выводим его с помощью метки `JLabel` внизу окна.

Конструктор класса `Dial` хранит минимальное, максимальное и текущее значения кругового регулятора; по умолчанию они равны 0, 100 и 0 соответственно. В конструкторе также устанавливается основной цвет компонента и регистрируются объекты-слушатели для перехвата событий, связанных с мышью. При нажатии клавиш мыши или перетаскивании указателя вызывается метод `spin()`, который обновляет текущее значение кругового регулятора. Этот метод также выполняет основные тригонометрические операции, чтобы определить, каким должно быть новое значение компонента.

Вся тригонометрия, связанная с отображением объекта `Dial`, находится в методах `paintComponent()` и `draw3DCircle()`. Частный вспомогательный метод `draw3DCircle()` рисует выпуклую или вдавленную окружность; это позволяет придать компоненту объемный вид.

Следующие несколько методов позволяют извлекать и изменять значения кругового регулятора (текущее, минимальное и максимальное). Здесь стоит обратить внимание на концепцию геттеров и сеттеров, которая применяется для всех важных полей класса `Dial`.

Мы обсудим ее подробней в главе 22. Также нужно остановиться на методе `setValue()`, который выполняет две важные задачи: перерисовывает компонент при поступлении нового значения и генерирует событие `DialEvent`, чтобы сообщить о внесенных изменениях.

В конце класса `Dial` находятся методы, необходимые для генерирования наших событий. Методы `addDialListener()` и `removeDialListener()` управляют списком объектов-слушателей.

Эта задача упрощается благодаря полю `listenerList`, унаследованному от класса `JComponent`. Для извлечения объектов-слушателей, связанных с текущим компонентом, используется метод `fireEvent()`; он отправляет объекты `DialEvent` всем зарегистрированным реализациям интерфейса `DialListener`.

Отделение модели данных от представления

Имейте в виду, что мы намеренно упростили наш пример. Все компоненты библиотеки Swing, которые мы рассматривали ранее, состоят из двух отдельных частей — модели данных и представления. Класс `Dial` объединяет эти составляющие в единое целое, что ограничивает возможность его повторного использования. Чтобы реализовать архи-

тектуру MVC, нам бы пришлось создать собственную модель данных и нечто вроде делегата пользовательского интерфейса, который бы занимался выводом компонента на экран и обрабатывал действия пользователя. Чтобы иметь полноценное представление об этом подходе, советуем вам ознакомиться с примером JogShuttle из книги «Java Swing» издательства O'Reilly.

В главе 19 мы закрепим пройденный материал. Вы научитесь использовать диспетчер компоновки для объединения в единое целое уже знакомых вам компонентов и контейнеров, и последующего создания на их основе сложных пользовательских интерфейсов.

Глава 19
ДИСПЕТЧЕРЫ КОМПОНОВКИ

Диспетчер компоновки занимается расстановкой дочерних компонентов внутри видимой области контейнера (см. рис. 19.1). Он контролирует их местоположение и размеры, учитывая определенные правила. Задача диспетчера компоновки состоит в размещении компонентов в рамках свободного пространства и соблюдении установленных между ними взаимоотношений. Библиотеки AWT и Swing предлагают несколько стандартных диспетчеров, которые покрывают собой большинство возможных ситуаций; но если у вас есть особые требования, вы можете создать собственный класс подобного рода.

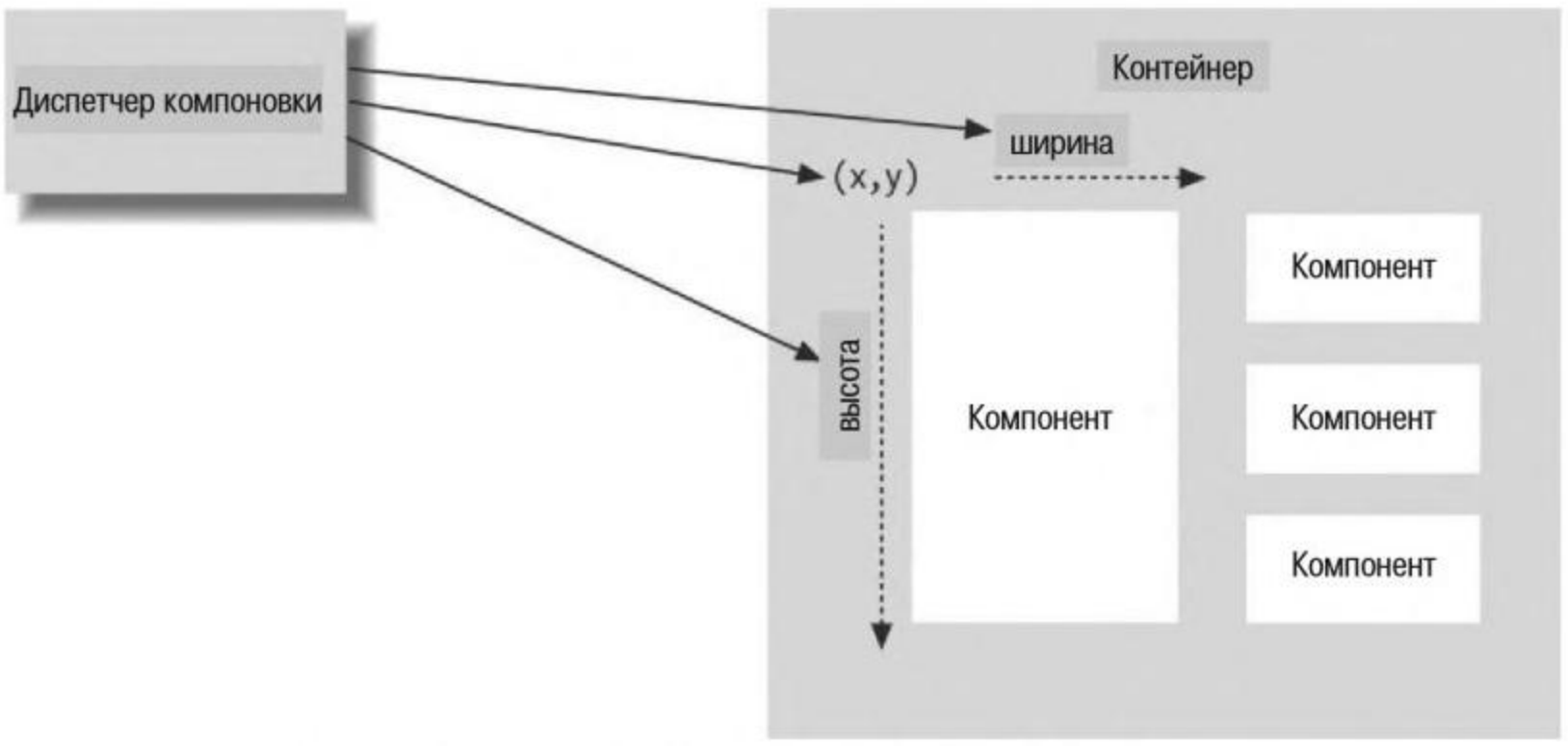

Рис. 19.1. Принцип работы диспетчера компоновки

У каждого контейнера есть свой стандартный диспетчер компоновки, в качестве которого выступает одна из разновидностей объекта `LayoutManager`. Но вы всегда можете установить свой собственный диспетчер, воспользовавшись методом `setLayout()`. Вот как это выглядит на примере объекта `BorderLayout`:

```
myContainer.setLayout( new BorderLayout() );
```

Обратите внимание, что хоть мы и создали объект `BorderLayout`, ссылка на него нигде не сохраняется. Это обычная ситуация; указанный

нами диспетчер компоновки делает свою работу незаметно, не афишируя свое взаимодействие с контейнером. Программисту не так уж часто приходится вызывать его методы напрямую, поэтому и ссылка на него обычно не сохраняется (исключением является объект `CardLayout`). Тем не менее при работе с компонентами вам нужно знать, какой диспетчер компоновки ими управляет.

Чтобы обновить свое содержимое, контейнер вызывает метод `doLayout()`, который, в свою очередь, обращается к объекту `LayoutManager`. Во время этого процесса из дочерних компонентов, размещенных внутри отображаемой области, вызываются методы `setLocation()` или `setBounds()`. Это происходит как при первом появлении контейнера на экране, так и в результате последующих вызовов из него метода `revalidate()`. Контейнеры, являющиеся производными класса `Window` (`Frame`, `JFrame` или `JWindow`), обновляются при любом масштабировании, в том числе и после вызова метода `pack()`, который подбирает для окна минимальные ширину и высоту, соблюдая предпочтительные размеры всех его компонентов.

Каждый компонент, который будет размещаться и масштабироваться диспетчером компоновки, обязан предоставить три важных значения: минимальный, максимальный и предпочтительный размеры. Для этого в классе `Component` предусмотрено три метода: `getMinimumSize()`, `getMaximumSize()` и `getPreferredSize()`. К примеру, кнопка `JButton` обычно масштабируется без всяких ограничений. Но дизайнер может указать для нее предпочтительный размер, при котором она будет выглядеть так, как ему хочется. В зависимости от установленных правил, диспетчер компоновки может либо проигнорировать это значение, либо использовать его, если ему больше не нужно учитывать никаких других требований. Когда у кнопки изменяется надпись, она должна заново вычислять свой минимальный размер, чтобы обеспечить корректное отображение нового текста. В целом, диспетчер компоновки должен соблюдать минимальный размер кнопки, гарантируя ей доступ к определенной области экрана. Точно так же у компонента могут возникнуть проблемы при отображении, если он станет слишком большим (возможно, ему придется увеличивать изображения, которые он выводит на экран); компонент может сообщить о максимально допустимом для себя размере при помощи метода `getMaximumSize()`.

Для контейнера предпочтительный размер имеет точно такой же смысл, как и для любого другого компонента. Но это значение контролируется не им самим, а его диспетчером компоновки, который отвечает за

размещение внутри него различных элементов управления. Таким образом, диспетчер компоновки подходит к решению этой проблемы с обеих сторон. Чтобы правильно расставить компоненты и контейнеры, он запрашивает у них их предпочтительные (или минимальные) размеры. Имея в своем распоряжении эти сведения, он рассчитывает предпочтительный размер для своего собственного контейнера (потом это значение может быть передано родительскому контейнеру на уровень выше и т. д.).

Диспетчер компоновки расставляет элементы управления внутри строго ограниченной области. Первым делом он обычно узнает габариты своего контейнера, а также предпочтительные (или минимальные) размеры его дочерних компонентов. После этого он отмеряет область экрана и устанавливает размеры подконтрольных ему объектов с учетом текущей схемой расстановки и определенных ограничениями.

Для задания минимального, максимального и предпочтительного размеров используются методы `setMinimumSize()`, `setMaximumSize()` и `setPreferredSize()`. Но будьте внимательны, когда их вызываете; вы должны учитывать реальное состояние компонента, а не указывать значения, которые подходят в какой-то одной конкретной ситуации. Вы можете переопределить методы `getMinimumSize()`, `getMaximumSize()` и `getPreferredSize()`, чтобы самостоятельно вычислять все нужные размеры, но это следует делать только в том случае, если к вашему компоненту выдвигаются какие-то дополнительные требования. В целом же, если вас не устраивает то, как масштабируются ваши элементы управления, то вы, скорее выбрали не тот диспетчер компоновки, или ваш пользовательский интерфейс плохо спроектирован. Часто бывает удобней использовать набор контейнеров `JPanel`, у каждого из которых есть свой экземпляр `LayoutManager`. Попробуйте разбить одну большую задачу на несколько мелких: размещайте связанные между собой компоненты на одной панели, и затем добавляйте полученные объекты в один общий контейнер. Если пользовательский интерфейс станет слишком громоздким, вы можете перейти на диспетчер компоновки, который накладывает строгие ограничения — например, `GridBagLayout` или `SpringLayout` (к ним мы еще вернемся в этой главе).

FlowLayout

`FlowLayout` — это простой диспетчер компоновки, который пытается разместить компоненты в контейнере слева направо и сверху вниз, учитывая их предпочтительные размеры. Вы можете указать для него

внутренние отступы по горизонтали и вертикали, а также способ выравнивания: LEFT, CENTER или RIGHT. По умолчанию все компоненты выравниваются по центру доступной области. FlowLayout является стандартным диспетчером компоновки для контейнера JPanel.

В следующем примере объект FlowLayout используется для добавления пяти кнопок на главную панель контейнера JFrame:

```
//файл: Flow.java
import javax.swing.*;
public class Flow extends JPanel {
   public Flow() {
      // FlowLayout является стандартным диспетчером
      // компоновки для контейнера JPanel
      add(new JButton("Один"));
      add(new JButton("Два"));
      add(new JButton("Три"));
      add(new JButton("Четыре"));
      add(new JButton("Пять"));
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("FlowLayout");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(400, 75);
      frame.setLocation(200, 200);
      Flow flow = new Flow();
      frame.setContentPane(flow);
      frame.setVisible(true);
   }
}
```

Результат показан на рис. 19.2.

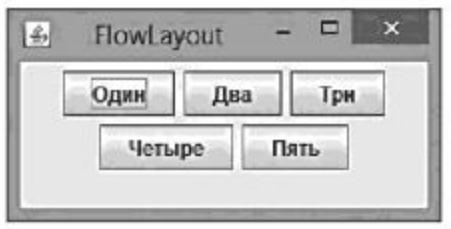

Рис. 19.2. Диспетчер компоновки FlowLayout

Попробуйте изменить размер окна. Когда оно станет достаточно узким, некоторые кнопки будут перенесены на вторую или третью строку.

GridLayout

Диспетчер компоновки под названием `GridLayout` делит контейнер на пропорциональные строки и столбцы. Компоненты масштабируются таким образом, чтобы заполнить собой отведенные им ячейки; соответственно, их минимальные и предпочтительные размеры игнорируются. Этот объект лучше всего подходит для расстановки компонентов с одинаковыми размерами; это может быть набор контейнеров `JPanel`, каждый из которых использует собственный диспетчер компоновки.

В конструкторе класса `GridLayout` можно указать количество строк и столбцов. Если компонентов станет слишком много, новые столбцы будут добавляться автоматически. Значение 0 сигнализирует о том, что вам все равно, сколько компонентов будет упаковано по соответствующему направлению (горизонтальному или вертикальному). Например, вызов `GridLayout(2,0)` формирует сетку с двумя строками и неограниченным количеством столбцов; если передать этому диспетчеру компоновки 10 элементов управления, получится две строки по пять столбцов в каждой.*

В следующем примере в качестве диспетчера компоновки используется объект `GridLayout`, который содержит три строки и два столбца:

```
//файл: Grid.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Grid extends JPanel {
   public Grid() {
   setLayout(new GridLayout(3, 2));
      add(new JButton("Один"));
      add(new JButton("Два"));
      add(new JButton("Три"));
      add(new JButton("Четыре"));
      add(new JButton("Пять"));
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("GridLayout");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

* Вызов `new GridLayout(0, 0)` приведет к исключению; один из параметров всегда должен быть больше ноля.

```
        frame.setSize(200, 200);
        frame.setLocation(200, 200);
        frame.setContentPane(new Grid());
        frame.setVisible(true);
    }
}
```

Результат показан на рис. 19.3.



Рис. 19.3. Диспетчер компоновки `GridLayout`

Пять кнопок размещены слева направо и сверху вниз, с одной пустой ячейкой.

BorderLayout

Диспетчер компоновки `BorderLayout` имеет одну интересную особенность. Он пытается распределить объекты по пяти «географическим» областям, которые представлены константами `NORTH` (север), `SOUTH` (юг), `EAST` (восток), `WEST` (запад) и `CENTER` (центр). Каждая из областей может иметь свой отступ. Этот диспетчер используется по умолчанию в главных панелях контейнеров `JWindow` и `JFrame`. В нем может храниться не больше пяти компонентов — по одному на каждое направление. Чтобы заполнить собой всю область, компоненты будут сжиматься или растягиваться. Это означает, что объекту `BorderLayout` лучше передавать целые контейнеры, а не отдельные элементы управления (в этом вы сможете убедиться, когда мы будем рассматривать наш второй пример).

Диспетчер компоновки `BorderLayout` требует, чтобы для компонента, который добавляется в контейнер, была указана определенная позиция. Для этого нужно использовать перегруженную версию метода `add()` из класса `Container`, которая позволяет указать *ограничение* в виде дополнительного аргумента. Ограничение представляет собой название области внутри объекта `BorderLayout`.

В следующем приложении мы устанавливаем для контейнера диспетчер компоновки BorderLayout, и уже традиционно добавляем в него пять кнопок. Каждая кнопка подписана в соответствии с названием области, в которой она находится:

```
//файл: Border1.java
import java.awt.*;
import javax.swing.*;
public class Border1 extends JPanel {
   public Border1() {
      setLayout(new BorderLayout( ));
      add(new JButton("Север"), BorderLayout.NORTH);
      add(new JButton("Юг"), BorderLayout.SOUTH);
      add(new JButton("Восток"), BorderLayout.EAST);
      add(new JButton("Запад"), BorderLayout.WEST);
      add(new JButton("Центр"), BorderLayout.CENTER);
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("BorderLayout1");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(300, 300);
      frame.setLocation(200, 200);
      frame.setContentPane(new Border1());
      frame.setVisible(true);
   }
}
```

Результат показан на рис. 19.4.

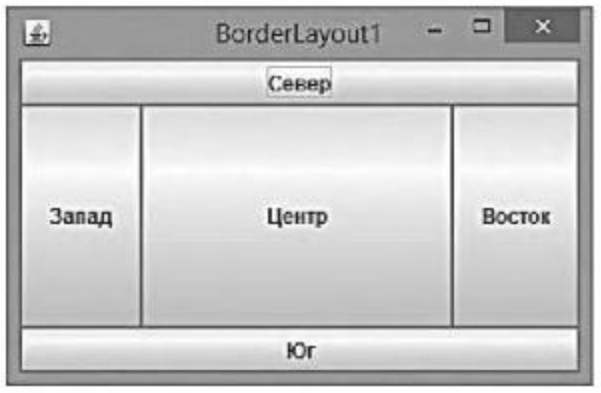

Рис. 19.4. Диспетчер компоновки BorderLayout

Как же на самом деле был поделен контейнер? Все просто: объекты в областях NORTH и SOUTH получили предпочтительную высоту, заполнив по горизонтали все свободное место; компонентам в областях EAST и WEST была присвоена предпочтительная ширина, а по вертикали они заняли

все то пространство, которое находится между «южным» и «северным» объектами. То, что осталось, отошло элементу управления в области CENTER. Как можно видеть на рис. 19.4, наши кнопки теперь имеют довольно причудливые пропорции. Но что если мы не хотим, чтобы диспетчер компоновки изменял размеры наших компонентов? Как вариант, можно поместить каждую кнопку в контейнер JPanel, который по умолчанию компонует свое содержимое с помощью объекта FlowLayout и, следовательно, учитывает предпочтительные размеры компонентов. Это означает, что кнопки не будут растягиваться вместе с окном. Этот подход проиллюстрирован в следующем приложении:

```
//файл: Border2.java
import java.awt.*;
import javax.swing.*;
public class Border2 extends JPanel {
   public Border2() {
      setLayout(new BorderLayout( ));
      JPanel p = new JPanel();
      p.add(new JButton("Север"));
      add(p, BorderLayout.NORTH);
      p =new JPanel();
      p.add(new JButton("Юг"));
      add(p, BorderLayout.SOUTH);
      p =new JPanel();
      p.add(new JButton("Восток"));
      add(p, BorderLayout.EAST);
      p =new JPanel();
      p.add(new JButton("Запад"));
      add(p, BorderLayout.WEST);
      p =new JPanel();
      p.add(new JButton("Центр"));
      add(p, BorderLayout.CENTER);
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("BorderLayout2");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(225, 150);
      frame.setLocation(200, 200);
      frame.setContentPane(new Border2());
      frame.setVisible(true);
   }
}
```

Результат показан на рис. 19.5.

В этом примере мы создали пять панелей, добавили в каждую из них по одной кнопке и разместили все это на окне, диспетчером компоновки для которого служит объект `BorderLayout`. Теперь все свободное пространство в области `CENTER` захватывает контейнер `JPanel`. Благодаря диспетчерам компоновки `FlowLayout` кнопки сохраняют свой предпочтительный размер и выравниваются по центру. Такой подход выглядит немного избыточным, поэтому очень скоро мы рассмотрим объект `GridBagLayout`, который позволяет сделать этот процесс более прозрачным.

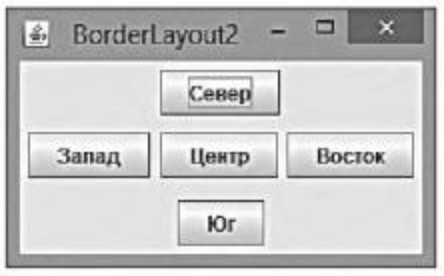

Рис. 19.5. Еще один пример использования диспетчера компоновки `BorderLayout`

BoxLayout

Большинство диспетчеров компоновки входят в состав пакета `java.awt` и берут свое начало еще с тех времен, когда выходила первая версия платформы Java. В пакете `javax.swing` содержится несколько новых диспетчеров общего назначения; одним из них является класс `BoxLayout`. Он размещает компоненты строго в одну линию (горизонтальную или вертикальную), что хорошо подходит для создания простых панелей инструментов. Этот диспетчер компоновки во многом похож на класс `FlowLayout`, только он не переносит элементы управления на новую строку.

Объект `BoxLayout` можно использовать напрямую, но библиотека Swing предоставляет удобный контейнер под названием `Box`, который отчасти автоматизирует этот процесс. Он содержит крайне полезные методы, которые помогут вам расставлять компоненты с помощью вышеупомянутого диспетчера компоновки.

Чтобы создать горизонтальную или вертикальную панель на основе класса `Box`, можно воспользоваться его статическими методами:

```
Container horizontalBox = Box.createHorizontalBox();
Container verticalBox = Box.createVerticalBox();
```

После этого можно добавлять компоненты с использованием привычного метода `add()`:

```
Container box = Box.createHorizontalBox();
box.add(new JButton("И говорит"));
```

Класс `Box` содержит еще несколько статических методов для создания специальных невидимых компонентов, которые могут использоваться для управления объектом `BoxLayout`. Первый из них, `createHorizontalGlue()` (и его вариация, `createHVerticalGlue()`), добавляет отступы между элементами управления. Когда контейнер изменяет свои размеры, отступы расширяются или сужаются с учетом доступного места. Есть еще одна пара специальных методов: `createHorizontalStrut()` и `createVerticalStrut()`; они делают практически то же самое, только отступы, созданные с их помощью, не меняют своих размеров.

Следующее приложение создает горизонтальный контейнер `Box` с двумя видами отступов между компонентами (см. рис. 19.6). Вы сами сможете увидеть, как изменение размеров окна влияет на расположение объектов.

```
//файл: Boxer.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Boxer extends JPanel {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Контейнер Box");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(250, 250);
      frame.setLocation(200, 200);
      Container box = Box.createHorizontalBox();
      box.add(Box.createHorizontalGlue());
      box.add(new JButton("И говорит"));
      box.add(Box.createHorizontalGlue());
      box.add(new JButton("ему"));
      box.add(Box.createHorizontalStrut(10));
      box.add(new JButton("щука"));
      box.add(Box.createHorizontalStrut(10));
      box.add(new JButton("человеческим"));
      box.add(Box.createHorizontalGlue());
      box.add(new JButton("голосом"));
```

```
        box.add(Box.createHorizontalGlue());
        frame.getContentPane().add(box, BorderLayout.CENTER);
        frame.pack();
        frame.setVisible(true);
    }
}
```



Рис. 19.6. Использование класса `Box`

Компоненты последовательно добавляются слева направо или сверху вниз; отступы между ними определяются с помощью методов `createHorizontalGlue()` и `createHorizontalStrut()`. По умолчанию компоненты размещаются один за другим без всяких промежутков. Метод `createHorizontalGlue()` играет роль своеобразной «пружины», позволяя смежным объектам равномерно занимать пространство контейнера. Метод `createHorizontalStrut()` устанавливает фиксированный отступ между соседними компонентами.

CardLayout

Диспетчер компоновки `CardLayout` специально предназначен для создания «стека» компонентов. Вместо того чтобы размещать на контейнере все объекты сразу, он выводит только один из них. С его помощью вы можете реализовать собственный вариант панели с вкладками. На самом деле, есть не так уж много задач, где может пригодиться этот диспетчер компоновки, особенно учитывая наличие готового компонента `JTabbedPane`, который был описан в главе 17. Мы рассматриваем его здесь в основном для полноты изложения.

Для добавления компонентов у объекта `CardLayout` есть своя версия метода `add()`, которая принимает два аргумента; первый из них служит в качестве строкового идентификатора для элемента стека.

```
add("netconfigscreen", myComponent);
```

Чтобы вынести определенный компонент на вершину стека, достаточно вызвать из объекта `CardLayout` метод `show()`, указав два аргумента: родительский контейнер и соответствующий элемент управления. Есть еще четыре метода для работы со стеком: `first()`, `last()`, `next()`

и previous(). Для их вызова вам понадобится ссылка на сам диспетчер компоновки, а не на его контейнер. Каждый из них принимает всего один аргумент — родительский объект Container. Ниже показан пример:

```
//файл: Card.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Card extends JPanel {
   CardLayout cards = new CardLayout();
   public Card() {
      setLayout(cards);
      ActionListener listener = new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            cards.next(Card.this);
         }
      };
      JButton button;
      button = new JButton("один");
      button.addActionListener(listener);
      add(button, "один");
      button = new JButton("два");
      button.addActionListener(listener);
      add(button, "два");
      button = new JButton("три");
      button.addActionListener(listener);
      add(button, "три");
}
   public static void main(String[] args) {
      JFrame frame = new JFrame("CardLayout");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(200, 200);
      frame.setLocation(200, 200);
      frame.setContentPane(new Card());
      frame.setVisible(true);
   }
}
```

Мы добавили в контейнер три кнопки, которые при нажатии сменяют друг друга. События, генерируемые кнопками, отслеживаются с помощью внутреннего анонимного класса; при этом из объекта CardLayout

каждый раз вызывается метод `next()`. Ссылка `Card.this` указывает на объект типа `Card`, который здесь выступает в качестве контейнера. В более реалистичном примере мы бы создали и передали экземпляру `CardLayout` набор панелей, каждая из которых могла бы содержать какую-то часть пользовательского интерфейса и имела бы свой отдельный диспетчер компоновки. Такие панели пытались бы занять собой все доступное место (то есть всю область контейнера, в котором они находятся), а их диспетчеры компоновки управляли бы размещением внутренних компонентов.

GridBagLayout

Класс `GridBagLayout` является крайне гибким диспетчером компоновки, который позволяет позиционировать компоненты друг относительно друга с учетом разных ограничений. С его помощью (приложив достаточное количество усилий) вы можете скомпоновать элементы интерфейса практически как угодно. Все объекты размещаются на виртуальной сетке согласно специальным логическим координатам. Мы называем их «логическими», потому что они обозначают позицию строк и столбцов, которые сформированы на основе размещенных внутри них компонентов. Ячейки этой виртуальной сетки растягиваются и сжимаются в зависимости от размеров элементов управления, которые в них хранятся, а также с учетом указанных ограничений.

Строки и столбцы объекта `GridBagLayout` масштабируются так, чтобы вместить самый крупный компонент (со всеми ограничениями, которые он накладывает). Любой компонент может занять сразу несколько строк или столбцов. Если элемент управления оказывается меньше своей ячейки, он может быть выровнен по любой ее стороне. Компоненты могут сохранять свой размер или расширяться в любом направлении. Дополнительные ячейки могут быть поделены в соответствии с «весом» своего содержимого (об этом чуть позже). Это позволяет управлять размерами разных компонентов при масштабировании окна.

Диспетчер компоновки `GridBagLayout` лучше всего использовать в сочетании с визуальным редактором пользовательских интерфейсов, иначе этот процесс будет напоминать охоту за слабым сигналом при помощи старой телевизионной антенны. Конечно, желаемого результата можно добиться и вручную, путем проб и ошибок, но при этом вы столкнетесь с отсутствием каких-либо простых и четких правил. `GridBagLayout` — довольно нетривиальный объект со своими нюансами.

К тому же, он лишен изящества как с точки зрения архитектуры, так и в плане реализации. Помните, что многие задачи можно решить с помощью вложенных контейнеров и комбинации более простых диспетчеров компоновки. В этой главе вы уже могли видеть примеры такого подхода. Вы сами должны решить, стоит ли вам переходить от простой компоновки к использованию сложных и многофункциональных объектов вроде `GridBagLayout`.

Класс `GridBagConstraints`

Несмотря на свою сложность и неприглядность, объект `GridBagLayout` имеет на удивление тривиальный API-интерфейс. Он предоставляет один конструктор без аргументов (`GridBagLayout()`) и всего лишь несколько методов, с помощью которых можно управлять отображением содержимого контейнера.

Внешний вид виртуальной сетки определяется набором объектов `GridBagConstraints` — с этого места начинаются странности. Каждый компонент, управляемый диспетчером компоновки `GridBagLayout`, связывается с отдельным экземпляром `GridBagConstraints`, в котором содержатся следующие переменные (чуть позже мы обсудим их подробней):

```
int gridx
int gridy
```

Определяют позицию компонента в виртуальной сетке

```
int weightx
int weighty
```

Отвечают за выделение дополнительного места в ячейке

```
int fill
```

Определяет, будет ли компонент расширяться, чтобы заполнить собой выделенное место

```
int gridheight
int gridwidth
```

Описывают количество строк или столбцов, которые должен занимать компонент

```
int anchor
```

Определяет позицию компонента в случае выделения дополнительного места

```
int ipadx
int ipady
```

Определяют расстояние между компонентом и границами его ячейки

```
Insets insets
```

Определяет расстояние между текущим компонентом и соседними элементами управления

Это то, что мы называем «ограничениями». Чтобы установить набор этих переменных для одного или нескольких компонентов, создайте экземпляр класса `GridBagConstraints` и передайте ему соответствующие значения (это можно сделать и с помощью конструктора, который принимает все 11 аргументов).

Для связывания набора ограничений с компонентом проще всего использовать версию метода `add()`, которая принимает как сам компонент, так и объект `GridBagConstraints`:

```
Container content = getContentPane();
JComponent component = new JLabel("ограничьте меня, пожалуйста...");
GridBagConstraints constraints = new GridBagConstraints();
constraints.gridx = x;
constraints.gridy = y;
...
content.add(component, constraints);
```

Но вы также можете использовать обычный метод `add()` с одним аргументом, устанавливая объект `GridBagConstraints` позже, с помощью метода `setConstraints()`:

```
add(component);
...
myGridBagLayout.setConstraints(component, constraints);
```

В любом случае, когда ограничения применяются к компоненту, происходит копирование. То есть вы можете изменять один и тот же экземпляр `GridBagConstraints`, передавая его разным объектам. Можно создать вспомогательный метод, который будет устанавливать нужный вам набор ограничений и передавать его диспетчеру компоновки вместе с элементом управления. Именно этот подход мы будем использовать в последующих примерах; наш вспомогательный метод под названием

addGB() будет принимать в качестве аргументов компонент и пару координат, которые потом будут превращаться в параметры gridx и gridy для объекта GridBagConstraints. Позже мы перегрузим этот метод и заставим его принимать аргументы для других ограничений, которые приходится часто устанавливать.

Координаты сетки

Один из самых неожиданных сюрпризов, которые преподносит объект GridBagLayout, заключается в том, что у нас нет возможности указать размер виртуальной сетки. Но так и было задумано. Количество строк и столбцов определяется косвенно, с учетом ограничений всех компонентов; диспетчер компоновки выбирает размеры, которых достаточно для того, чтобы вместить все объекты. Например, если передав диспетчеру один компонент и установив для него параметры gridx и gridy, каждый из которых равен 25, мы получим виртуальную сетку размером 25×25, в которой строки и столбцы имеют диапазон от 0 до 24. Если добавить второй компонент, у которого параметры gridx и gridy равны, соответственно, 30 и 13, то виртуальная сетка будет иметь размерность 30×25. Вам не нужно беспокоиться о вычислении необходимого количества строк и столбцов. Диспетчер компоновки делает это автоматически по мере добавления компонентов.

Теперь, когда мы знаем, как работает объект GridBagLayout, можно перейти к небольшому примеру. Для начала разместим группу кнопок в форме креста. При добавлении компонентов в виртуальную сетку мы будем указывать локальные переменные x и y. Отчасти это делается для большей наглядности, но это также может пригодиться в том случае, если нужно будет разместить набор компонентов в определенном ряду, горизонтальном или вертикальном.

Перед добавлением каждого следующего объекта достаточно инкрементировать значения параметров gridx или gridy. Это позволяет легко и без лишних проблем позиционировать элементы управления друг относительно друга (позже мы узнаем, как автоматизировать этот процесс с помощью константы RELATIVE из класса GridBagConstraints). Итак, ниже представлен код для нашего первого примера (см. рис. 19.7):

```
//файл: GridBag1.java
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
public class GridBag1 extends JPanel {
   GridBagConstraints constraints = new GridBagConstraints();
   public GridBag1() {
      setLayout(new GridBagLayout());
      int x, y; // для наглядности
      addGB(new JButton("Север"), x = 1, y = 0);
      addGB(new JButton("Запад"), x = 0, y = 1);
      addGB(new JButton("Центр"), x = 1, y = 1);
      addGB(new JButton("Восток"), x = 2, y = 1);
      addGB(new JButton("Юг"), x = 1, y = 2);
   }
   void addGB(Component component, int x, int y) {
      constraints.gridx = x;
      constraints.gridy = y;
      add(component, constraints);
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("GridBagLayout1");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(225, 150);
      frame.setLocation(200, 200);
      frame.setContentPane(new GridBag1());
      frame.setVisible(true);
   }
}
```

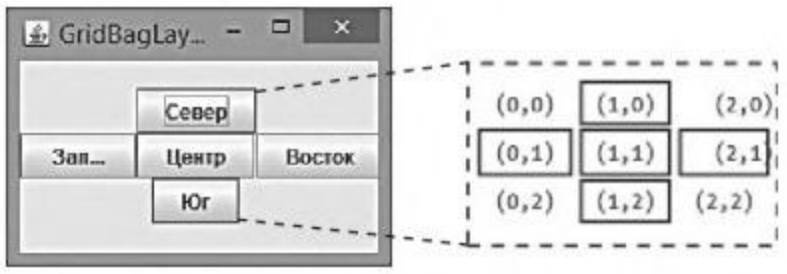

Рис. 19.7. Простой пример использования объекта `GridBagLayout`

Все кнопки в этом примере собраны в центре области отображения; они не заполняют собой все доступное место, так как контейнер соблюдает их предпочтительные размеры.

Именно так будет вести себя данный диспетчер компоновки, если не указать ни одного параметра вида «`weight*`». Больше об этих ограничениях мы поговорим в следующих двух разделах.

Параметр fill

Давайте сделаем так, чтобы кнопки занимали все пространство окна `JFrame`.

Для этого нужно сделать две вещи: указать для ограничения `fill` константу `BOTH` и установить параметрам `weightx` и `weighty` ненулевые значения:

```
//файл: GridBag2.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GridBag2 extends JPanel {
   GridBagConstraints constraints = new GridBagConstraints();

   public GridBag2() {
      setLayout(new GridBagLayout());
      constraints.weightx = 1.0;
      constraints.weighty = 1.0;
      constraints.fill = GridBagConstraints.BOTH;
      int x, y; // для наглядности
      addGB(new JButton("Север"), x = 1, y = 0);
      addGB(new JButton("Запад"), x = 0, y = 1);
      addGB(new JButton("Центр"), x = 1, y = 1);
      addGB(new JButton("Восток"), x = 2, y = 1);
      addGB(new JButton("Юг"), x = 1, y = 2);
   }
   void addGB(Component component, int x, int y) {
      constraints.gridx = x;
      constraints.gridy = y;
      add(component, constraints);
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("GridBagLayout2");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(225, 150);
      frame.setLocation(200, 200);
      frame.setContentPane(new GridBag2());
      frame.setVisible(true);
   }
}
```

Итоговое размещение компонентов показано на рис. 19.8.

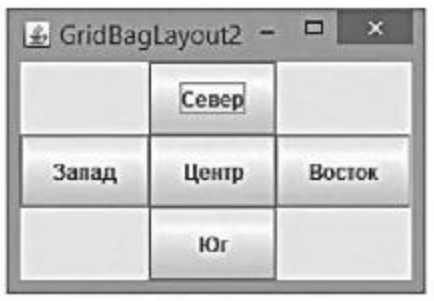

Рис. 19.8. Делаем так, чтобы кнопки заполнили собой все доступное место

Константа `BOTH` содержится в классе `GridBagConstraints`; она разрешает компоненту занимать все доступное пространство по горизонтали и вертикали. Вот какие константы можно присваивать параметру `fill`:

`HORIZONTAL`

Заполнение всего доступного места по горизонтали.

`VERTICAL`

Заполнение всего доступного места по вертикали.

`BOTH`

Заполнение всего доступного места в обоих направлениях.

`NONE`

Запрет на заполнение доступного места; используются предпочтительные размеры компонентов.

В этом примере параметрам `weightx` и `weighty` нужно было присвоить одно и то же ненулевое значение, неважно какое именно; мы выбрали `1,0`. По умолчанию эти параметры равны нолю, поэтому раньше наши компоненты не заполняли собой свободное место.

Заполнение нескольких строк или столбцов

Одна из самых важных возможностей диспетчера компоновки `GridBaglayout` заключается в том, что он позволяет занимать одному компоненту сразу несколько строк или столбцов. Для этого нужно установить определенные значения для параметров `gridwidth` и `gridheight` из объекта `GridBagConstraints`. Именно такая компоновка элементов управления выполняется в следующем примере; первая кнопка занима-

ет две ячейки по вертикали, а четвертой отводится две ячейки по горизонтали. Результат показан на рис. 19.9.

```
//файл: GridBag3.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GridBag3 extends JPanel {
   GridBagConstraints constraints = new GridBagConstraints();
   public GridBag3() {
      setLayout(new GridBagLayout());
      constraints.weightx = 1.0;
      constraints.weighty = 1.0;
      constraints.fill = GridBagConstraints.BOTH;
      int x, y; // для наглядности
      constraints.gridheight = 2; // занимает две строки
      addGB(new JButton("один"), x = 0, y = 0);
      // возвращаем значение по умолчанию
      constraints.gridheight = 1;
      addGB(new JButton("два"), x = 1, y = 0);
      addGB(new JButton("три"), x = 2, y = 0);
      constraints.gridwidth = 2; // занимает два столбца
      addGB(new JButton("четыре"), x = 1, y = 1);
      // возвращаем значение по умолчанию
      constraints.gridwidth = 1;
   }
   void addGB(Component component, int x, int y) {
      constraints.gridx = x;
      constraints.gridy = y;
      add(component, constraints);
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("GridBagLayout3");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(200, 200);
      frame.setLocation(200, 200);
      frame.setContentPane(new GridBag3());
      frame.setVisible(true);
   }
}
```

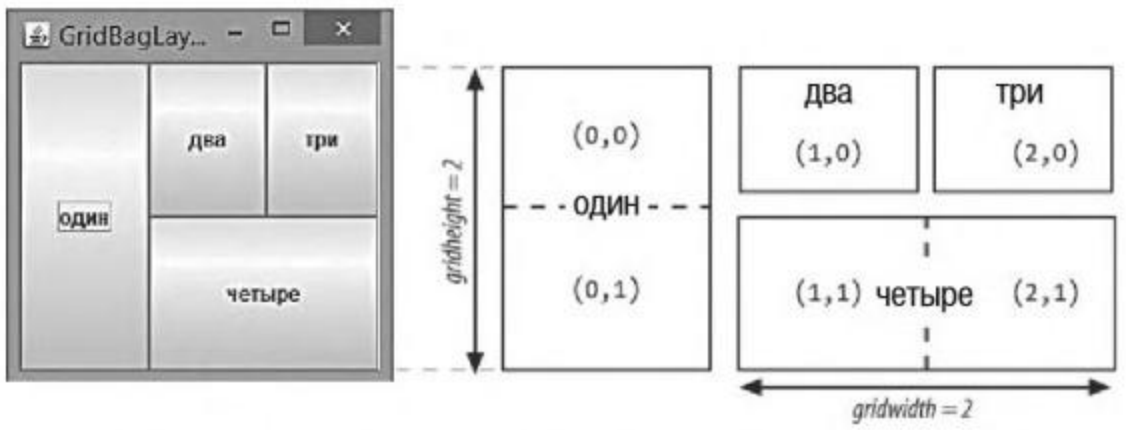

Рис. 19.9. Делаем так, чтобы компоненты занимали сразу две ячейки

Размер каждого элемента определяется значениями параметров `gridwidth` и `gridheight` из объекта `GridBagConstraints`. У первой кнопки параметр `gridheight` равен 2; это означает, что она занимает две строки. Ее виртуальные координаты `gridx` и `gridy` равны 0, поэтому ей отводятся две соседние ячейки — (0,0) и (0,1). Согласно тому же принципу, четвертая кнопка занимает два соседних столбца, (1,1) и (2,1), так как ее параметр `gridwidth` равен 2.

В этом примере параметры `fill`, `weightx` и `weighty` имеют одинаковые значения для всех компонентов; они равны `BOTH`, 1 и 1 соответственно. Благодаря этому все кнопки имеют одинаковый «вес» и занимают все доступное им место. Так вам легче будет понять, сколько пространства отводится каждому компоненту, хотя, строго говоря, эти параметры не являются обязательными.

Параметры `weightx` и `weighty`

Поля `weightx` и `weighty` объекта `GridBagConstraints` определяют так называемый «вес» компонентов, согласно которому распределяется дополнительное место в контейнере. В простых ситуациях эти параметры имеют довольно понятный эффект: чем больший у компонента «вес», тем больше места ему будет отведено по сравнению с его соседями. На рис. 19.10 показано, как влияет значение параметра `weightx` на три кнопки, размещенные в один ряд.

Вот код этого примера:

```
//файл: GridBag4.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GridBag4 extends JPanel {
```

```
GridBagConstraints constraints = new GridBagConstraints();
public GridBag4() {
    setLayout(new GridBagLayout());
    constraints.fill = GridBagConstraints.BOTH;
    constraints.weighty = 1.0;
    int x, y; // для наглядности
    constraints.weightx = 0.1;
    addGB(new JButton("один"), x = 0, y = 0);
    constraints.weightx = 0.5;
    addGB(new JButton("два"), ++x, y);
    constraints.weightx = 1.0;
    addGB(new JButton("три"), ++x, y);
  }
  void addGB(Component component, int x, int y) {
    constraints.gridx = x;
    constraints.gridy = y;
    add(component, constraints);
  }
  public static void main(String[] args) {
    JFrame frame = new JFrame("GridBagLayout4");
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setSize(300, 100);
    frame.setLocation(200, 200);
    frame.setContentPane(new GridBag4());
    frame.setVisible(true);
  }
}
```



Рис. 19.10. Управление размером компонентов с помощью параметра `weightx`

Сами по себе значения параметров `weightx` и `weighty` не имеют никакого смысла; важны их относительные пропорции. Получив сведения о предпочтительных размерах компонентов (включая внутренние и внешние отступы, о которых мы поговорим в следующем разделе), диспетчер компоновки выделяет им дополнительное место пропорционально их «весу». К примеру, если бы все компоненты имели одинако-

вый «вес», каждый из них получил бы треть от доступного пространства. Чтобы сделать этот процесс более наглядным, «вес» для строк и столбцов лучше представлять в виде дробных чисел, общая сумма которых равно `1,0`. Например: `0,25`, `0,25`, `0,50`. Компоненту с нулевым «весом» дополнительное место не выделяется.

Все становится немного сложнее, когда приходится иметь дело с несколькими строками и столбцами и когда один компонент может занимать сразу несколько ячеек. Обычно объект `GridBagLayout` сначала определяет общий» вес для всех строк и столбцов, и только потом пропорционально распределяет между ними дополнительное пространство. Стоит заметить, что наш предыдущий пример является всего лишь частным случаем, когда в каждом столбце находится по одному компоненту. Ниже вы можете ознакомиться со всеми подробностями этих вычислений.

Вычисление «веса» строк и столбцов

Когда объект `GridBagLayout` анализирует определенный ряд ячеек (горизонтальный или вертикальный), он учитывает «вес» каждого компонента, который размещен в этом ряду; при этом компоненты, которые занимают несколько ячеек, игнорируются. В качестве «веса» строки или столбца берется самое большое значение параметра `weightx/weighty` из имеющихся. Как видите, диспетчер компоновки `GridBagLayout` пытается приспосабливаться к нуждам самого «тяжелого» компонента в заданном ряду.

После этого объект `GridBagLayout` начинает рассматривать компоненты, которые занимают больше одной ячейки. Здесь-то и появляются настоящие странности. Диспетчер компоновки пытается понять, влияют ли эти компоненты на определение самого «тяжелого» элемента в строке или столбце. Но так как они занимают сразу несколько ячеек, их «вес» распределяется между всеми рядами, к которым они принадлежат.

Объект `GridBagLayout` пытается вычислить фактический «вес» отдельных частей компонента, которому отводится несколько ячеек. Для этого он распределяет значение `weightx/weighty` между рядами, соблюдая пропорции, согласно которым компонент разделен по ширине/высоте. Но откуда он берет эти пропорции, если виртуальная сетка еще не до конца определена? Оказывается, диспетчер компоновки пользуется гипотетическими значениями, вычисленными на основе уже имеющего-

ся «веса» строк и столбцов. С их помощью он распределяет «вес» между рядами, которые занимает компонент. Полученный результат используется в качестве «веса» компонента для определенного ряда и сравнивается с наибольшим значением параметра `weightx/weighty` в строке или столбце. В итоге, как мы уже говорили выше, общий вес ряда может измениться.

Параметр `anchor`

По умолчанию, если компонент не занимает все доступное ему место, он выводится по центру. Но это всего лишь один из вариантов. Чтобы указать объекту `GridBagLayout`, как нужно выравнивать компонент внутри ячейки, используется параметр `anchor`. Допускаются следующие значения: `GridBagConstraints.CENTER`, `NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST` и `NORTHWEST`. Например, константа `GridBagConstraints.NORTH` выполняет выравнивание по верхнему краю, а значение `SOUTHEAST` размещает элемент управления в правом нижнем углу.

Внутренние и внешние отступы

Управлять поведением компонентов внутри виртуальной сетки диспетчера `GridBagLayout` можно также с помощью внутренних и внешних отступов. Внутренние отступы определяются полями `ipadx` и `ipady` из объекта `GridBagConstraints`. Они описывают так называемые горизонтальные и вертикальные «факторы роста» компонента. Взгляните на рис. 19.11. Кнопка **Запад** больше остальных, потому что ее параметры `ipadx` и `ipady` равны 25. Это означает, что диспетчер компоновки берет предпочтительный размер компонента и добавляет к нему 25 точек в каждом направлении — это и есть та область, которую занимает кнопка. Остальные компоненты имеют нулевой отступ (значение по умолчанию), поэтому их размер не изменился. Но обратите внимание на промежутки между ними. Кнопка **Запад** слишком уж высокая, из-за чего средняя строка в сетке должна быть выше остальных.

```
//файл: GridBag5.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GridBag5 extends JPanel {
   GridBagConstraints constraints = new GridBagConstraints();
```

```
    public GridBag5() {
        setLayout(new GridBagLayout());
        int x, y; // для наглядности
        addGB(new JButton("Север"), x = 1, y = 0);
        constraints.ipadx = 25; // добавляем отступ
        constraints.ipady = 25;
        addGB(new JButton("Запад"), x = 0, y = 1);
        constraints.ipadx = 0; // удаляем отступ
        constraints.ipady = 0;
        addGB(new JButton("Центр"), x = 1, y = 1);
        addGB(new JButton("Восток"), x = 2, y = 1);
        addGB(new JButton("Юг"), x = 1, y = 2);
    }
    void addGB(Component component, int x, int y) {
        constraints.gridx = x;
        constraints.gridy = y;
        add(component, constraints);
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridBagLayout5");
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize(250, 250);
        frame.setLocation(200, 200);
        frame.setContentPane(new GridBag5());
        frame.setVisible(true);
    }
}
```

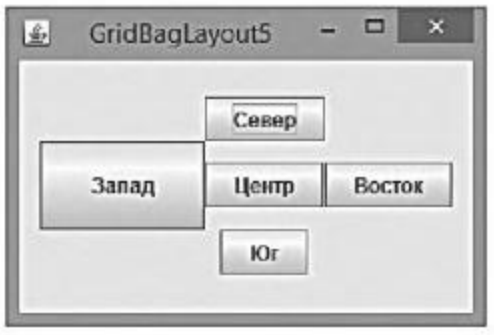

Рис. 19.11. Компоновка с использованием внутренних и внешних отступов

Обратите внимание, что внутренний отступ по горизонтали, `ipadx`, добавляется с обеих сторон кнопки.

Таким образом, ширина кнопки увеличивается на значение, которое в два раза больше параметра `ipadx`. То же самое касается отступа по вертикали, `ipady` — он добавляется как сверху, так и снизу.

Внешние отступы — это пространство между краями компонента и его ячейкой. Они определяются значением типа `Insets`, которое хранится в поле `insets` класса `GridBagConstraints`. Объект `Insets`, в свою очередь, содержит еще четыре поля, которые описывают отступы по всем четырем краям: `top`, `bottom`, `left` и `right`. Во всех этих параметрах можно легко запутаться. Если внутренний отступ добавляется к самому компоненту, увеличивая его размер, то поле `insets` описывает расстояние между компонентом и его ячейкой. Это проиллюстрировано на рис. 19.12.

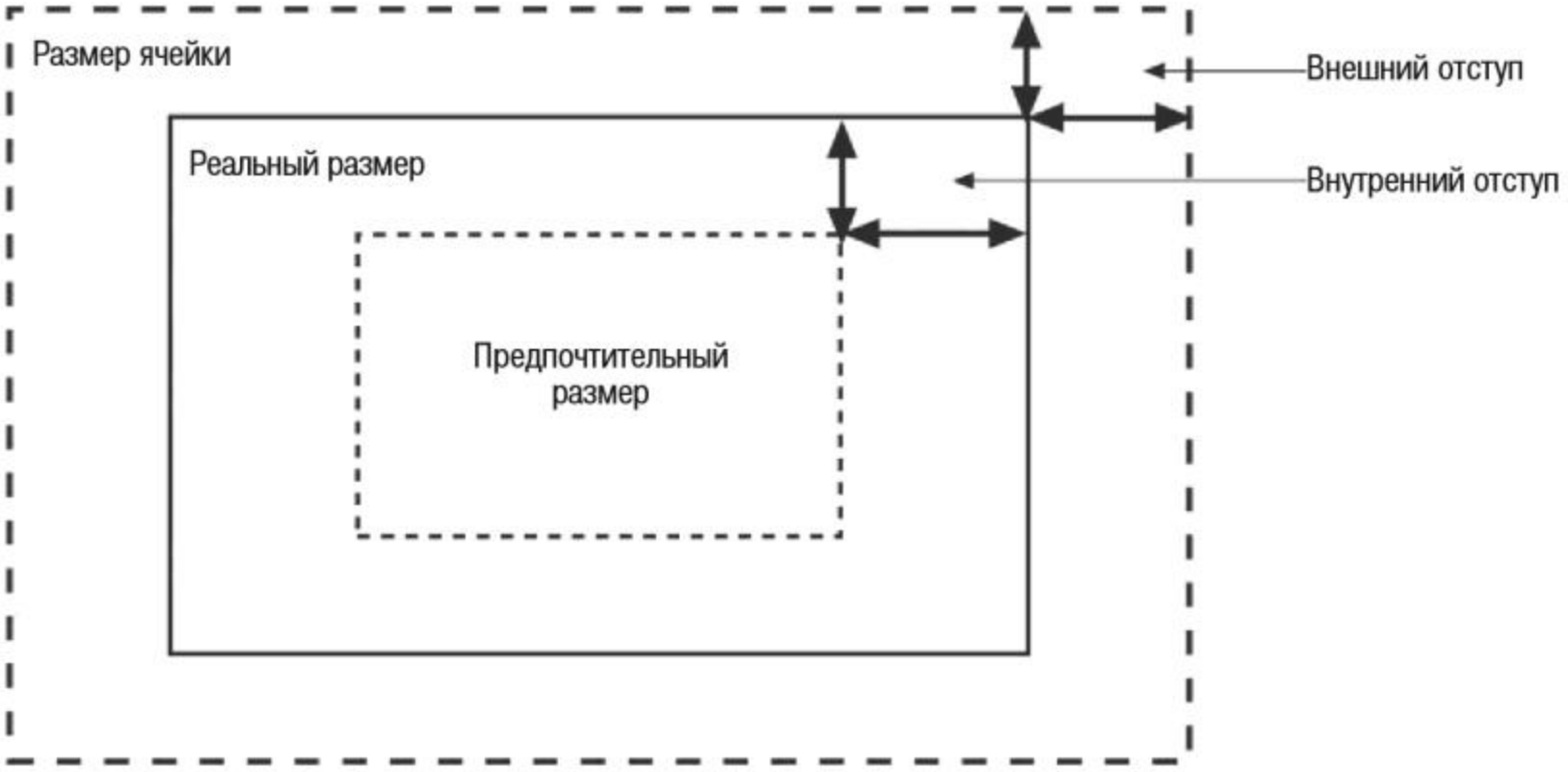

Рис. 19.12. Разница между внутренними и внешними отступами

Внутренние отступы и «вес» компонента ведут себя довольно странно в сочетании друг с другом. Если вы решили изменить стандартные параметры `ipadx/ipady`, вам лучше оставить значения `weightx/weighty` по умолчанию.

Относительное позиционирование

Пока что при использовании диспетчера компоновки `GridBagLayout` нам неизменно приходилось вручную указывать ограничения `gridx` и `gridy` для каждого компонента. Но существует и другой подход.

В данном разделе речь пойдет об *относительном позиционировании*. Концептуально в этом нет ничего сложного; мы просто будем размещать один компонент относительно другого (например, справа или снизу). Для этого вы можете присвоить параметрам `gridx` или `gridy` константу

GridBagConstraints.RELATIVE. К сожалению, здесь тоже не обходится без некоторых неочевидных моментов:

- Чтобы разместить компонент справа от предыдущего элемента управления, нужно присвоить параметру gridx константу RELATIVE *и* передать параметру gridy значение, которое использовалось для предыдущего компонента.
- Аналогично, чтобы разместить компонент снизу от предыдущего, необходимо присвоить параметру gridy константу RELATIVE *и* оставить значение gridx неизменным.
- Если все компоненты будут иметь параметры gridx и gridy со значением RELATIVE, они разместятся в один ряд, а не по диагонали, как вы могли бы ожидать (это поведение по умолчанию).

Другими словами, если одному из параметров, gridx или gridy, присвоена константа RELATIVE, вам лучше оставить все остальные значения без изменений. Относительное позиционирование позволяет легко размещать множество компонентов в один ряд (по горизонтали или вертикали). Именно для этого оно и предназначено; в остальных случаях его использование превратит вашу работу с диспетчером компоновки в настоящую борьбу.

Объект GridBagLayout поддерживает еще один вид относительного позиционирования, который подразумевает использование полей gridwidth и gridheight из класса GridBagConstraints. С их помощью можно указать, где должен размещаться компонент — в строке или столбце. Если передать одному из этих параметров константу REMAINDER, компонент будет размещен в конце ряда и, как следствие, должен будет занять собой все оставшееся место. Если вы хотите сделать компонент предпоследним, используйте константу RELATIVE. К сожалению, это позволяет описывать ограничения, которые нельзя соблюсти на практике; например, вы можете потребовать, чтобы сразу два компонента были «последними» в столбце, чего не может быть в реальности. В этом случае диспетчер компоновки пытается сделать что-то логичное, но в результате вы практически наверняка получите совсем не то, чего ожидали.

Совместное использование разных диспетчеров компоновки

В позиционировании компонентов, как и в реальной жизни, не все и не всегда проходит гладко. Например, если мы захотим использовать

возможности объекта GridBagLayout только для некоторых компонентов, нам придется создавать отдельные диспетчеры компоновки для разных частей пользовательского интерфейса и затем объединять их вместе. Именно так мы сформировали интерфейс карманного калькулятора, представленного на рис. 19.13. Для этого нам понадобилось три экземпляра GridBagLayout: один для первого ряда кнопок (C, %, +), один для последнего (0, ., =) и еще один для самого окна. Главный диспетчер компоновки (тот, что принадлежит контейнеру JFrame) содержит поле ввода, которое мы используем в качестве экрана калькулятора, панели с верхним и нижним рядами кнопок, а также 12 кнопок посередине*.

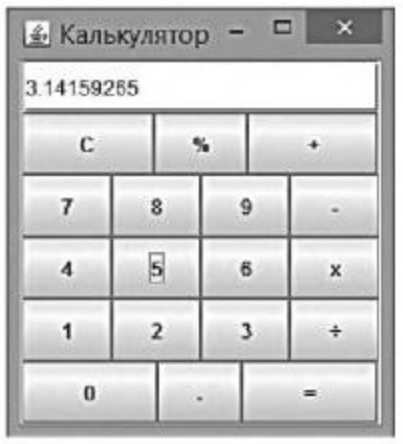

Рис. 19.13. Виртуальный калькулятор

Код этого калькулятора, представленный ниже, реализует только пользовательский интерфейс (то есть клавиши и экран). Все, что вы набираете, будет отображаться вверху; для очистки экрана используется кнопка **C** (clear — очистить). Код, который относится к связыванию пользовательского интерфейса с настоящими математическими операциями, мы предлагаем вам реализовать самостоятельно. На правах совета: при каждом нажатии кнопки со знаком равенства вы можете отправлять событие в объект, который занимается вычислениями; тот, в свою очередь, сможет извлечь содержимое поля ввода, выполнить расчет и вывести результат.

```
//файл: Calculator.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Calculator extendsJPanel implements ActionListener {
   GridBagConstraints gbc = new GridBagConstraints();
```

* Если вам интересно, прототипом этого калькулятора является модель ELORG-801, на которую автор наткнулся на веб-сайте **www.taswegian.com/MOSCOWx/elorg801.html**

```
JTextField theDisplay = new JTextField();
public Calculator() {
   gbc.weightx  = 1.0;gbc.weighty = 1.0;
   gbc.fill = GridBagConstraints.BOTH;
   ContainerListener listener = new ContainerAdapter() {
      public void componentAdded(ContainerEvent   e) {
Component comp = e.getChild();
if (comp instanceof JButton)
   ((JButton)comp).addActionListener(Calculator.this);
      }
   };
   addContainerListener(listener);
gbc.gridwidth = 4;
addGB(this, theDisplay, 0, 0);
// создаем верхний ряд кнопок
JPanel topRow = new   JPanel(  );
topRow.addContainerListener(listener);
gbc.gridwidth = 1;
gbc.weightx  = 1.0;
addGB(topRow, new JButton("C"), 0, 0);
gbc.weightx  = 0.33;
addGB(topRow, new JButton("%"), 1, 0);
gbc.weightx  = 1.0;
addGB(topRow, new JButton("+"), 2, 0 );
gbc.gridwidth = 4;
addGB(this, topRow,  0, 1);
gbc.weightx  = 1.0;gbc.gridwidth = 1;
// создаем кнопки с цифрами
   for(int j=0; j<3; j++)
for(int  i=0; i<3; i++)
      addGB(this, new JButton("" + ((2-j)*3+i+1) ), i, j+2);
   // -, x, и деление
   addGB(this, new JButton("-"), 3, 2);
   addGB(this, new JButton("x"), 3, 3);
   addGB(this, new JButton("\u00F7"), 3, 4);
   // создаем нижний ряд кнопок
   JPanel bottomRow = new JPanel();
   bottomRow.addContainerListener(listener);
   gbc.weightx  = 1.0;
   addGB(bottomRow, new JButton("0"), 0, 0);
   gbc.weightx  = 0.33;
```

```
        addGB(bottomRow, new JButton("."), 1, 0);
        gbc.weightx  = 1.0;
        addGB(bottomRow, new JButton("="), 2, 0);
        gbc.gridwidth = 4;
        addGB(this, bottomRow, 0, 5);
    }
    void addGB(Container cont, Component comp, int x, int   y) {
        if ((cont.getLayout() instanceof GridBagLayout) ==   false)
            cont.setLayout(new GridBagLayout( ));
        gbc.gridx =  x; gbc.gridy = y;
        cont.add(comp, gbc);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("C"))
            theDisplay.setText("");
        else
            theDisplay.setText(theDisplay.getText()
             + e.getActionCommand());
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame("Калькулятор");
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize(200, 250);
        frame.setLocation(200, 200);
        frame.setContentPane(new Calculator());
        frame.setVisible(true);
    }
}
```

Чтобы передать объекту `GridBagLayout` компоненты с определенными ограничениями, мы опять воспользовались вспомогательным методом `addGB()`. Давайте рассмотрим его подробней, прежде чем переходить непосредственно к расстановке элементов управления. Выше мы упоминали, что наш интерфейс состоит из трех диспетчеров компоновки: один для окна приложения, а остальные для двух рядов кнопок — верхнего (`topRow`) и нижнего (`bottomRow`). Метод `addGB()` используется во всех трех случаях; в качестве первого его аргумента выступает контейнер, в который добавляются компоненты. Таким образом, передавая ему ссылку **this**, мы размещаем элемент управления прямо на главной панели объекта `JFrame`. Чтобы добавить кнопку в верхний ряд, мы передаем объект `topRow`. Первым делом метод `addGB()` проверяет диспетчер

компоновки полученного контейнера и при необходимости приводит его к типу `GridBagLayout`. Затем он устанавливает позицию компонента, изменяя набор его ограничений, хранящихся в объекте `gbc`, и передает результат контейнеру.

В этом примере мы используем всего один набор ограничений, изменяя его параметры по своему усмотрению. Объект `GridBagConstraints` инициализируется в конструкторе класса `Calculator`. Перед вызовом метода `addGB()` мы изменяем те его поля, начальные значения которых нам не подходят. Таким образом, мы добавили поле ввода прямо на главную панель окна **(this)**, установив для него сетку шириной в четыре ячейки. При добавлении каждого нового компонента метод `addGB()` создает копию объекта `GridBagConstraints`, поэтому во время работы приложения нам достаточно всего одной переменной `gbc`.

Для двух рядов кнопок (верхнего и нижнего) мы решили использовать два разных диспетчера компоновки типа `GridBagLayout`. Мы не могли обойтись одной общей сеткой, несмотря на то, что кнопки находятся на одной линии. Таким образом, у контейнера `topRow` есть свой собственный диспетчер компоновки с тремя столбцами — по одному на каждый компонент. Все ячейки имеют одинарную ширину.

Чтобы контролировать размеры компонентов, мы устанавливаем для них параметры `weightx`; это позволяет сделать крайние кнопки **(C** и **+)** больше, чем среднюю **(%)**. Затем мы добавляем в приложение весь контейнер `topRow`, выделяя ему ширину в 4 ячейки. Нижний ряд создается похожим образом.

Для создания кнопок с цифрами от 1 до 9 мы используем двойной вложенный цикл. Эта конструкция выглядит довольно изящно, но больше о ней сказать нечего. С операциями вычитания, умножения и деления тоже все просто: мы создаем кнопки с соответствующими надписями и добавляем их в окно приложения с помощью метода `addGB()`. Стоит отметить, что для вывода настоящего знака деления мы использовали не обычный прямой слеш, а специальную константу из набора символов Unicode.

На этом построение пользовательского интерфейса закончено; осталось только организовать обработку событий. Каждая кнопка генерирует объекты типа `ActionEvent`, поэтому мы должны зарегистрировать для них реализации интерфейсов слушателей. В качестве слушателя для всех кнопок у нас будет выступать главная панель окна — `Calculator`. При регистрации объекта `Calculator` мы проявим определенную сме-

калку. Когда компонент добавляется на панель, контейнер генерирует событие `ContainerEvent`. Для регистрации слушателей наших кнопок мы используем внутреннюю анонимную реализацию `ContainerListener`. Это означает, что объект `Calculator` должен зарегистрироваться в качестве `ContainerListener` как для самого себя, так и для двух панелей, `topRow` и `bottomRow`. Метод `componentAdded()` работает крайне просто: чтобы понять, какой компонент сгенерировал событие (то есть, какой объект был добавлен), он делает вызов `getChild()`, и если это кнопка, то в качестве реализации интерфейса `ActionListener` он регистрирует для нее объект `Calculator`.

Метод `actionPerformed()` вызывается при нажатии любой кнопки и добавляет в поле ввода соответствующую команду действия (которая совпадает с меткой); исключение составляет кнопка **C**, которая удаляет все то, что было набрано ранее.

Объединение нескольких диспетчеров компоновки — это очень полезный прием. Конечно, наш пример в какой-то степени избыточен. Не так уж часто приходится использовать сразу три объекта `GridBagLayout`. Однако с диспетчером `BorderLayout` все обстоит совсем иначе; он позволяет использовать в своих областях разные контейнеры, и эта возможность довольно востребована. Например, в области `CENTER` может находиться контейнер `ScrollPane`, у которого есть свой специальный диспетчер компоновки, а области `EAST` и `SOUTH` при необходимости могут хранить панели с поддержкой диспетчеров `GridBagLayout` или `FlowLayout`.

Другие диспетчеры компоновки

Мы рассмотрели наиболее распространенные диспетчеры компоновки; с их помощью можно создать практически любой пользовательский интерфейс. Но в пакете `javax.swing` осталось еще два класса, о которых нужно упомянуть — `SpringLayout` и `GroupLayout`. Они были созданы специально для визуальных редакторов графических интерфейсов, но их также можно использовать и в программном коде. Подробности об этих диспетчерах компоновки и их сторонних аналогах, таких как `MigLayout`* и `FormLayout`** от компании JGoodies, вы можете найти в книге «Java Swing» издательства O'Reilly.

* **www.miglayout.com**
** **www.jgoodies.com**

Абсолютное позиционирование

Можно и вовсе обойтись без диспетчера компоновки, если передать в метод `setLayout()` значение `null`. Обычно это делается для того, чтобы позиционировать компоненты в абсолютных координатах. Но это плохая практика. Минимальные размеры элементов управления могут отличаться от платформы к платформе, поэтому с точки зрения переносимости ваш пользовательский интерфейс многое потеряет.

В следующем примере вместо диспетчера компоновки используются абсолютные координаты:

```
//файл: MoveButton.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MoveButton extends JPanel {
   JButton button = new JButton("Я двигаюсь");
   public MoveButton() {
      setLayout(null);
      add(button);
      button.setSize(button.getPreferredSize());
      button.setLocation(20, 20);
      addMouseListener(new MouseAdapter() {
         public void mousePressed(MouseEvent e) {
            button.setLocation(e.getX(), e.getY());
         }
      });
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame("Передвижная кнопка");
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      frame.setSize(250, 200);
      frame.setLocation(200, 200);
      frame.setContentPane(new MoveButton());
      frame.setVisible(true);
   }
}
```

Чтобы передвинуть кнопку на новое место, щелкните мышью за ее пределами. При изменении размеров окна кнопка всегда будет оставаться на одном и том же месте относительно его левого верхнего угла.

Глава 20

РИСОВАНИЕ С ИСПОЛЬЗОВАНИЕМ 2D API

В предыдущих главах мы рассмотрели компоненты и архитектуру компонентов графических пользовательских интерфейсов (graphical user interfaces, GUIs) используя пакеты Java Swing и AWT. Теперь мы перейдем на уровень ниже и рассмотрим процедурные API для визуализации графики в приложениях Java. Эти API используются компонентами для рисования на экране, для отображения значков и графики. Как и Java7, Java2D поддерживается на всех основных платформах, то есть хорошо построенные Java-клиенты должны быть такими же по производительности, как и native-приложения

Эта глава посвящена некоторым деталям языка Java, касающимся сложностей 2D, зависимости от разрешения API, предназначенных для рисования, и инструментов ядра для загрузки и отображения изображений. В следующей главе мы исследуем инструменты динамической обработки изображений более детально и рассмотрим классы, позволяющие создавать и изменять обрабатываемые изображения пиксел за пикселом.

Общая картина

Классы, которые вы будете использовать для рисования, содержатся в шести пакетах: `java.awt`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.image`, и `java.awt.print`. Совместно, эти классы составляют большую часть 2D API и охватывают рисование форм, изображений и текста. На рис. 20.1 представлен «вид с высоты птичьего полета» на эти классы. В 2D API содержится намного больше, чем мы можем охватить в двух главах. Чтобы ближе познакомиться с ним, прочитайте книгу «Java 2D Graphics» Джоната Кнадсена (O'Reilly).

Экземпляр объекта класса `java.awt.Graphics2D` называется *графическим контекстом*. Он представляет поверхность рисования как область

отображения компонента, страницу печати или внеэкранный буфер изображения. Графический контекст поддерживает методы для рисования трех типов графических объектов: форм, изображений и текста. Объект класса `Graphics2D` называется графическим контекстом потому, что он дополнительно содержит контекстную информацию об области отображения. Эта информация включает отсеченный участок области отображения, цвет заливки, способ передачи данных, шрифт текста и геометрическую трансформацию. Если представить область отображения как холст художника, то графический контекст — это мольберт, который содержит набор инструментов и отметок на рабочей области.

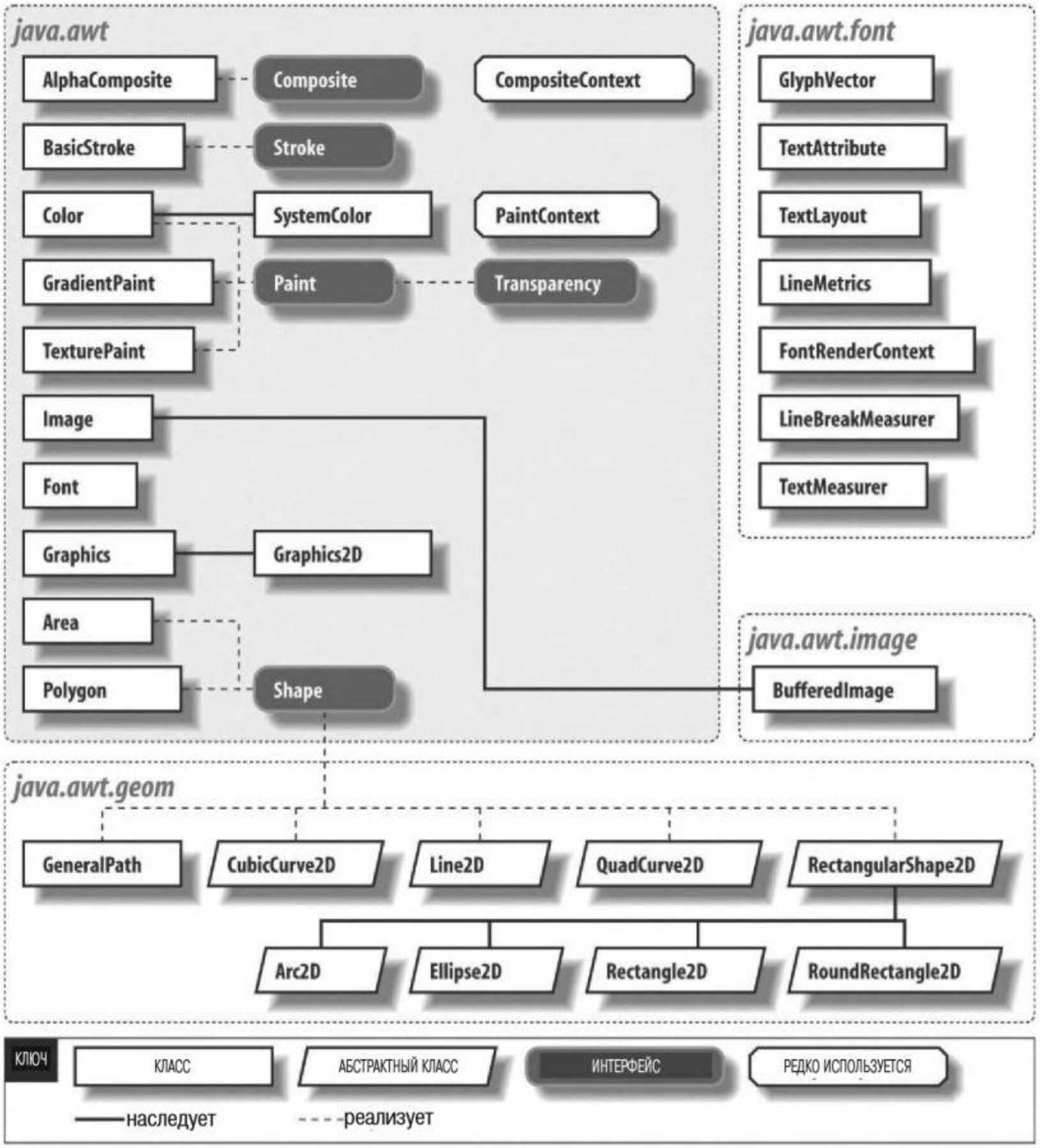

Рис. 20.1. Графические классы 2D API

Существует четыре способа получить объект класса `Graphics2D`. В следующем списке они представлены в порядке от более общего способа к менее общему:

Из библиотек AWT или Swing, как результат графического запроса к компоненту

В этом случае новый графический контекст для соответствующей области создается и передается вашему компоненту методами `paint()` или `update()`. (На самом деле, метод `update()` применим только к компонентам AWT, но не к новым компонентам Swing)

Прямым запросом к внеэкранному буферу изображения

В этом случае мы запрашиваем графический контекст напрямую из внеэкранного буфера. Мы будем использовать такой подход при рассмотрении техники двойной буферизации.

Копированием существующего объекта класса Graphics2D

Дублирование графического объекта может быть полезным при выполнении более сложных операций рисования; разные копии объекта класса `Graphics2D` могут быть нарисованы в одной области, имея различные атрибуты и отсеченные области. Объект класса `Graphics2D` можно скопировать, вызвав метод `create()`.

Напрямую с экранного компонента

Существует возможность запросить объект с принадлежащей ему области отображения. Однако это практически всегда приводит к ошибке. Если вы испытываете соблазн сделать это, подумайте, зачем вам изобретать велосипед, вместо того чтобы использовать обычный механизм методов `paint()` и `repaint()`.

Каждый раз при вызове метода `paint()` оконная система создает компонент нового объекта класса `Graphics2D` для рисования в области отображения. Это означает, что атрибуты, установленные в одной сессии рисования: цвет рисования или область отсечения будут переустановлены при следующем вызове метода `paint()` (каждый вызов метода начинается с нового чистого холста). Для большинства основных атрибутов: цвет первого плана, цвет фона, шрифт мы можем установить значения по умолчанию для конкретного компонента. В дальнейшем при рисовании в данном компоненте графический контекст будет соответствовать свойствам, установленным при инициализации.

Метод `paint()` может исключить возможность доступа к тому, что уже нарисовано на экране. Он отвечает за визуализацию всей рабочей области. Высокоуровневые API обычно отвечают за буферизацию вывода и ограничение количества вызовов метода `paint()` для компонента. Компоненты AWT дополнительно могут использовать метод `update()`, который позволяет им обновлять их вид, предполагается, что их предыдущая работа все еще на экране. Однако этот метод не используется компонентами Swing.

Для обратной совместимости графический контекст всегда передается методу `paint()` как объект типа `Graphics`. Если вы хотите воспользоваться замечательными функциями 2D API — а вы наверняка хотите, — вам необходимо направить эту ссылку к объекту класса `Graphics2D`. Вы увидите, как это работает в следующих примерах.

Конвейер визуализации

Одно из преимуществ 2D API заключается в том, что формы, изображения и текст управляются во многом аналогичными способами. В этой части мы обсудим, что происходит с формами, изображениями и текстом после того, как вы передаете их объекту класса `Graphics2D`. *Визуализация* — это процесс получения некоторой коллекции форм (графических примитивов), изображений и текста и выяснения способа их представления в виде цветных пикселов на экране или при печати. Класс `Graphics2D` поддерживает четыре операции визуализации:

- Рисование контура форм методом `draw()` Заливка внутренней части формы методом `fill()`
- Рисование произвольного текста методом `drawString()`
- Рисование изображения любой из форм методом `drawImage()`.

Графический контекст представлен как объект класса `Graphics2D` содержащий следующие свойства, значения которых контролируются соответствующими методами доступа (например, `getFont()` и `setFont()`):

`Paint`

Текущее значение свойства *paint* (объект типа `java.awt.Paint`) определяет, какой цвет или шаблон будет использоваться для заливки формы. Это свойство одинаково влияет и на рисование контура формы, и на рисование текста. Вы можете изменить текущее значение свой-

ства paint, используя метод setPaint() класса Graphics2D. Отметим, что класс Color реализует интерфейс Paint, таким образом, вы можете передать объект класса Colors в метод setPaint(), если хотите использовать сплошные цвета.

Stroke

Класс Graphics2D использует свойство stroke, чтобы определить, как рисовать контур формы, которая передана ему методом draw(). В терминах графики, «штриховать» форму, значит прочертить путь определенный формой, используя карандаш или кисть заданного размера и характеристик. Например, рисуя форму круга с использованием штриха, который выглядит как сплошная линия, мы получим форму кольца или шайбы. Объект stroke в Graphics2D API — немного более абстрактное понятие. Он получает форму для создания штриха и возвращает замкнутый контур формы, который далее будет заполнен методами класса Graphics2D. Вы можете установить текущее значение свойства stroke, используя метод setStroke(). В 2D API есть удобный класс java.awt.BasicStroke, который реализует линии различной ширины, стилей соединений и концов линий и пунктиров.

Transformation

Формы, изображения и текст могут быть геометрически трансформированы, перед визуализацией. Это значит, что они могут быть перемещены, повернуты и растянуты. Трансформация, реализованная классом Graphics2D, переводит координаты из «пространства пользователя» в «пространство устройства». По умолчанию, класс Graphics2D использует трансформацию, которая приводит 72 единицы в пространстве пользователя к одному дюйму в устройстве вывода. Если вы рисуете линию из точки (0,0) в точку (72,0), используя трансформацию по умолчанию, линия будет 1 дюйм длиной, независимо от того, нарисована она на экране или на странице печати. Текущая трансформация может быть изменена методами translate(), rotate(), scale(), и shear().

Composing rule

Правило композиции определяет, как цвет новой операции рисования будет совмещен с уже существующим цветом на поверхности рисования объекта Graphics2D. Атрибуты, установленные методом setComposite(), получают экземпляр java.awt.AlphaComposite. Композиция позволяет вам сделать части изображения или рисунка полностью или частично прозрачными или сочетать их другими интересными способами.

`Clipping shape`

Все операции визуализации ограничены внутренней областью отсеченного участка изображения. Ни один пиксел за пределами этой формы не будет изменен. По умолчанию, область отсечения позволяет рисовать на всей поверхности рисования (обычно прямоугольная область `Component`). Однако в дальнейшем вы можете ограничить эту область любой простой или сложной формой (например, формой текста)

`Rendering hints`

Существуют разные методики, которые могут быть использованы для вывода графических примитивов. Обычно приходится искать компромисс между скоростью и качеством визуализации. Настройки визуализации (константа, определенная в классе `RenderingHints`) определяют, какая техника используется.

Графические примитивы (формы, текст и изображения) передаются через устройство визуализации в результате серии операций, называемых конвейером визуализации. Пройдемся по конвейеру. Его можно разделить на четыре шага, первый шаг зависит от операции визуализации:

- Трансформация формы. Для форм, которые будут закрашены, форма просто трансформируется, с использованием текущей трансформации `Graphics2D`. Для форм, чей контур нарисован методом `draw()`, он заштриховывается с применением текущего штриха. Символы текста преобразуются в формы, при этом используется текущий шрифт. Итоговая форма текста трансформируется подобно любой другой закрашенной форме. Изображение также использует текущие настройки трансформации.
- Определение используемых цветов. Для заливки области текущий объект `paint` определяет цвета, которые необходимо использовать при закрашивании формы. При рисовании изображения, цвета берутся из самого изображения.
- Совмещение цветов с существующими на поверхности рисования, с использованием текущих настроек правила композиции.
- Отсечение результата, с использованием текущих настроек свойств области отсечения.

Настройки визуализации используются для управления качеством отрисовки.

Краткий тур по Java 2D

Далее мы отправляемся в краткий тур по Java 2D, включающий работу с формами и текстом. Этот раздел мы завершим примером, иллюстрирующим Java 2D в действии.

Заполнение форм

Простейший способ использования конвейера визуализации заключается в заполнении форм. Например, следующий код создает эллипс и закрашивает его сплошным цветом. Этот код будет работать внутри метода `paint()`. Полноценный, готовый к выполнению пример мы представим несколько позже.

```
Shape c = new Ellipse2D.Float(50, 25, 150, 150); // x,y,ширина, высота
g2.setPaint(Color.blue);
g2.fill(c);
```

Здесь `g2` — это наш объект `Graphics2D`. Абстрактный класс формы `Ellipse2D`, но реализуемый конкретными внутренними подклассами `Float` и `Double`, которые работают со значениями вещественного и двойной точности типа, соответственно. Аналогично класс `Rectangle2D` обладает конкретными классами `Rectangle2D.Float` и `Rectangle2D.Double`.

Рисование контура формы

Рисование контура формы немного сложнее. Взгляните на следующий пример:

```
Shape r = new Rectangle2D.Float(100, 75, 100, 100);
g2.setStroke(new BasicStroke(4));
g2.setPaint(Color.yellow);
g2.draw(r);
```

Здесь мы говорим объекту `Graphics2D`, использовать штрих четыре единицы в ширину сплошного желтого цвета. Когда мы вызываем метод `draw()`, объект `Graphics2D` использует указанный штрих для создания нового контура данного прямоугольника. Далее форма контура закрашивается как в предыдущем примере; так рисуется контур прямоугольника. Сам прямоугольник не закрашивается.

Удобные методы

Пакет `Graphics2D` включает довольно много удобных методов для рисования и закрашивания стандартных форм, эти методы фактически наследуют класс `Graphics`. В табл. 20.1 кратко описаны эти методы. Немного проще вызвать метод `fillRect()`, чем создавать экземпляр формы прямоугольника и передавать его методу `fill()`.

Табл. 20.1. Методы рисования форм в графическом классе

Метод	Описание
`draw3DRect()`	Рисует подсвеченный 3D прямоугольник
`drawArc()`	Рисует угол
`drawLine()`	Рисует линию
`drawOval()`	Рисует овал
`drawPolygon()`	Рисует многоугольник, замыкая его соединением конечных точек
`drawPolyline()`	Рисует линию соединяющую серию точек без замыкания
`drawRect()`	Рисует прямоугольник
`drawRoundRect()`	Рисует прямоугольник со скругленными углами
`fill3DRect()`	Рисует закрашенный подсвеченный трехмерный прямоугольник
`fillArc()`	Рисует закрашенный угол
`fillOval()`	Рисует закрашенный овал
`fillPolygon()`	Рисует закрашенный многоугольник
`fillRect()`	Рисует закрашенный прямоугольник
`FillRoundRect()`	Рисует закрашенный прямоугольник со скругленными углами

Как видите, для каждого метода `fill()` в таблице, есть соответствующий метод `draw()`, который визуализирует форму, как при рисовании линий без закраски. Исключая методы `fillArc()` и `fillPoligon()`, каждый метод берет только значения координат `x` и `y` для определения верхнего левого угла формы и `width` и `height` для определения ее размера.

Наиболее гибкий метод рисует многоугольник, который определяется двумя массивами, содержащими координаты вершин `x` и `y`. Методы класса `Graphics` получают два таких массива и рисуют контур многоугольника или закрашенный многоугольник.

Методы перечисленные в Табл. 20.1 — это быстрый доступ к более общим методам `Graphics2D`. Более важные процедуры это, для начала, создание объекта `java.awt.geom.Shape` и затем передача его методам

draw() и fill() пакета Graphics2D. Например, вы могли бы создать объект Polygon из массива координат. Так как Polygon реализует интерфейс Shape, вы можете передать его главным методам пакета Graphics2D: draw() и fill().

Метод fillArc() запрашивает шесть целочисленных аргументов. Первые четыре определяют границы поля для овала так же, как метод fillOval(). Последние два аргумента определяют, какой участок овала мы хотим нарисовать, как начальную позицию угла и смещение, оба аргумента указываются в градусах. Отметка ноль градусов в направлении на три часа, положительное значение угла по часовой стрелке. Например, для рисования правой половины круга, мы должны вызвать:

```
g.fillArc(0, 0, radius * 2, radius * 2, -90, 180);
```

Метод draw3DRect() автоматически выбирает цвет тени, «затемняя» текущий цвет. То есть вы должны установить какой-нибудь цвет (возможно серый или белый), отличный от черного , используемого по умолчанию. Иначе вы получите черный прямоугольник с толстым контуром.

Рисование текста

Как и рисование контуров формы, рисование текста является простой вариацией закраски формы. Когда вы просите Graphics2D нарисовать текст, он определяет форму, которую необходимо нарисовать и затем закрасить. Шрифт — это коллекция начертаний. Пример рисования текста:

```
g2.setFont(new Font("Times New Roman", Font.PLAIN, 64));
g2.setPaint(Color.red);
g2.drawString("Hello, 2D!", 50, 150);
```

Когда мы вызываем метод drawString(), пакет Graphics2D использует текущий шрифт для извлечения начертаний, соответствующих символам в строке. Далее начертания, которые в действительности просто объекты Shape, закрашиваются, с использованием текущего значения свойства Paint.

Рисование изображений

Изображения обрабатываются несколько иначе, чем формы. В частности текущее значение свойства Paint не используется при визуализа-

ции изображения, так как изображение содержит собственную информацию о цвете для каждого пиксела (это и есть `paint` на самом деле). Следующий пример загружает изображение из файла и отображает его:

```
Image i = Toolkit.getDefaultToolkit().getImage("camel.gif");
g2.drawImage(i, 75, 50, this);
```

В этом примере вызов метода `drawImage()` сообщает `Graphics2D` разместить изображение в заданном положении. Позже мы объясним значение четвертого аргумента, использующегося для мониторинга загрузки изображения.

Трансформации и визуализация

Конвейер состоит из четырех стадий, которые влияют на каждую графическую операцию. В частности, все визуализируемое подлежит трансформации, композиции и выделению. Настройки визуализации используются для влияния на всю визуализацию класса `Graphics2D`.

Этот пример показывает, как изменить трансформацию, используя перенос и вращение:

```
g2.translate(50, 0);
g2.rotate(Math.PI / 6);
```

К каждому графическому примитиву рисуемому объектом `g2` теперь будут применены указанные трансформации: сдвиг на 50 единиц вправо и вращение на 30 градусов по часовой стрелке. Мы можем достичь похожего глобального эффекта для композиции изображений:

```
AlphaComposite ac = AlphaComposite.getInstance(
AlphaComposite.SRC_OVER, (float).5);
g2.setComposite(ac);
```

Теперь каждый графический примитив, нарисованный нами, будет полупрозрачным; позже мы подробнее объясним этот момент.

Все операции рисования отсекаются за текущей областью отсечения, которой является любой объект реализующий интерфейс `Shape`.

В следующем примере область отсечения принимает форму эллипса:

```
Shape e = new Ellipse2D.Float(50, 25, 250, 150);
g2.clip(e);
```

Вы можете получить текущую форму отсечения, используя метод `getclip()`, что удобно, если вы хотите восстановить ее позже, используя метод `setClip()`.

Наконец настройки визуализации влияют на все операции рисования. В следующем примере, мы говорим объекту класса `Graphics2D` использовать сглаживание, технику, которая сглаживает неровности краевых пикселов форм или текста:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
```

Класс `RenderingHints` содержит другие ключи и значения, которые представляют другие настройки визуализации. Если вы правда любите покрутить ручки на приборной доске, то этот класс будет вам полезен.

Пример целиком

Давайте наконец объединим все, хотя бы для того, чтобы показать, как графические примитивы совместно продвигаются по конвейеру визуализации. Следующий пример демонстрирует применение класса `Graphics2D` от начала и до конца конвейера визуализации. Используя минимальное количество кода, мы можем нарисовать симпатичную завершенную композицию (см. рис. 20.2)

Ниже представлен код:

```
//файл: Iguana.java
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
public class Iguana extends JComponent {
   private Image image;
   private int theta;
   public Iguana() {
      image = Toolkit.getDefaultToolkit().getImage(
         "Piazza di Spagna.small.jpg");
      theta = 0;
      addMouseListener(new MouseAdapter() {
         public void mousePressed(MouseEvent me) {
            theta = (theta + 15) % 360;
```

```
                repaint();
            }
        });
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        int cx = getSize().width / 2;
        int cy = getSize().height / 2;
        g2.translate(cx, cy);
        g2.rotate(theta * Math.PI / 180);
        Shape oldClip = g2.getClip();
        Shape e = new Ellipse2D.Float(-cx, -cy, cx * 2, cy * 2);
        g2.clip(e);
        Shape c = new Ellipse2D.Float(-cx, -cy, cx * 3 / 4, cy * 2);
        g2.setPaint(new GradientPaint(40, 40, Color.blue,
            60, 50, Color.white, true));
        g2.fill(c);
        g2.setPaint(Color.yellow);
        g2.fillOval(cx / 4, 0, cx, cy);
        g2.setClip(oldClip);
        g2.setFont(new Font("Times New Roman", Font.PLAIN, 64));
        g2.setPaint(new GradientPaint(-cx, 0, Color.red,
            cx, 0, Color.black, false));
        g2.drawString("Hello, 2D!", -cx * 3 / 4, cy / 4);
        AlphaComposite ac = AlphaComposite.getInstance(
            AlphaComposite.SRC_OVER, (float).75);
        g2.setComposite(ac);
        Shape r = new RoundRectangle2D.Float(0, -cy * 3 / 4,
            cx * 3 / 4, cy * 3 / 4, 20, 20);
        g2.setStroke(new BasicStroke(4));
        g2.setPaint(Color.magenta);
        g2.fill(r);
        g2.setPaint(Color.green);
        g2.draw(r);
        g2.drawImage(image, -cx / 2, -cy / 2, this);
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame("Iguana");
        frame.setLayout(new BorderLayout());
```

```
        frame.add(new Iguana(), BorderLayout.CENTER);
        frame.setSize(300, 300);
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible(true);
    }
}
```



Рис. 20.2. Использование 2D API

Класс `Iguana` — это наследник класса `JComponent` с весьма причудливым методом `paint()`. В методе `main()` создается объект `JFrame`, который содержит компонент `Iguana`.

Конструктор класса `Iguana` загружает небольшое изображение (позже мы расскажем об этом подробнее) и настраивает обработчик событий мыши. Этот обработчик изменяет переменную экземпляра `theta` и перерисовывает компонент. Каждый раз, когда вы нажимаете кнопку мыши, все изображение поворачивается на 15 градусов.

Метод `paint()` класса `Iguana` выполняет довольно интересные действия, но ни одно из них не является слишком сложным. Во-первых, пространство пользователя трансформируется таким образом, что начало изображения располагается в центре компонента. Пространство пользователя поворачивается на угол `theta`:

```
g2.translate(cx, cy);
g2.rotate(theta * Math.PI / 180);
```

Класс `Iguana` сохраняет текущую (по умолчанию) форму отсечения до превращения ее в большой эллипс. Далее класс Iguana рисует два закрашенных эллипса. Первый рисуется экземпляром класса `Ellipse2D` и затем закрашивается. Второй рисуется с помощью удобного метода `fillOval()`. Мы расскажем о цветовом градиенте первого эллипса в следующей части. Как вы можете увидеть на рис. 20.2, оба эллипса отсечены эллиптической формой отсечения. После закрашивания двух эллипсов, класс `Iguana` возвращает предыдущую форму отсечения.

Далее, класс `Iguana` рисует текст (см. раздел «Использование шрифтов» далее). Дальнейшее действие: изменить правило композиции следующим образом:

```
AlphaComposite ac = AlphaComposite.getInstance(
AlphaComposite.SRC_OVER, (float).75);
g2.setComposite(ac);
```

Здесь важно только одно: мы хотим, чтобы все было нарисовано с применением прозрачности. Класс `AlphaComposite` определяет константы, предоставляющие различные предопределенные цвета. В этом случае мы запрашиваем правило *источник над назначением* (`SRC_OVER`), с дополнительным множителем альфа, равным 0,75. Источник над назначением означает, что все, что мы рисуем (источник) должно быть отображено над тем, что уже нарисовано (назначение). Множитель альфа означает, что у всего, что мы нарисуем, вместо обычного значения полупрозрачности будет установлено значение 0,75 или три четверти, позволяя существующему изображению «просвечивать» сквозь нарисованное позже.

Вы можете увидеть эффект нового правила композиции на скругленном прямоугольнике и изображении, они оба выглядят полупрозрачными.

Закрашивание форм

Класс `Iguana` закрашивает собственные формы набором цветов, используя метод `setPaint()` класса `Graphics2D`. Этот метод устанавливает текущий цвет в графическом контексте, то есть мы настраиваем его перед каждой операцией рисования. Метод `setPaint()` влияет на любой объект, реализующий интерфейс `Paint`. 2D API включает три реализации этого интерфейса, представляющих сплошные цвета, цветовой градиент и текстуры.

Сплошные цвета

Класс `java.awt.Color` представляет цвет в Java. Объект класса `Color` описывает отдельный цвет и реализует интерфейс `Paint`, чтобы закрасить этим цветом область. Вы можете создать сложный объект `Color`, указав целые значения красного, зеленого и синего, от 0 до 255, или значения с плавающей точкой от 0.0 до 1.0. Немного странный метод

`getColor()` может использовать поиск по имени цвета в таблице системных настроек, как описано в главе 11.

Кроме того класс `Color` определяет набор `static final` значений цвета; мы использовали их в примере `Iguana`. Константы, такие как `Color.black` и `Color.red`, предоставляют соответствующий набор базовых объектов цвета для рисования.



Чрезмерное создание резервных экземпляров цвета является распространенной причиной нехватки памяти Java клиентов. Рассмотрите возможность использования шаблонов, чтобы быть уверенными, что вы не создали 200 экземпляров небесно-голубого.

Цветовой градиент

Градиент цвета — это сглаженное смешение двух или более цветов. Класс `GradientPaint` воплощает эту идею в удобной реализации интерфейса `Paint`. Все, что вам нужно сделать — определить две точки и цвет в каждой из них. О таких деталях, как плавный переход цвета из одной точки в другую, позаботится класс `GradientPaint`. В предыдущем примере эллипс закрашивался градиентом следующим образом:

```
g2.setPaint(new GradientPaint(40, 40, Color.blue,
60, 50, Color.white, true));
```

Последний параметр в конструкторе `GradientPaint` определяет, будет ли градиент *циклическим*. В циклическом градиенте цвет сохраняет колебания за пределами двух точек, которые вы указали. В противном случае, градиент рисует переход цвета только от одной точки до другой. За конечными точками цвет сплошной.

В Java 6 добавлена возможность многоостановочного градиента в `LinearGradientPaint` и `RadialGradientPaint`. Многоостановочный градиент может, например, создавать сглаженный переход от зеленого к голубому и красному.

Текстуры

Текстуры — это просто изображения, повторяющиеся снова и снова, как напольная плитка. В 2D API эта концепция представлена в классе

TexturePaint. Для создания текстуры достаточно указать исходное изображение и прямоугольник, который будет использован для его воспроизведения. Для этого вы также должны знать, как создавать и использовать изображения, что мы обсудим немного позже.

Системная палитра

Класс Color упрощает создание определенного цвета, однако это не всегда то, что вы хотите сделать. Иногда вы хотите получить предопределенную цветовую схему. Это крайне важно, когда вы разрабатываете пользовательский интерфейс; желательно, чтобы ваши компоненты имели те же цвета, что и другие компоненты платформы, и автоматически изменялись, если пользователь изменит свою цветовую схему. Для этого и нужен класс SystemColor. Системная палитра представляет цвета, используемые локальной оконной системой в конкретном контексте. Класс SystemColor содержит множество предопределенных системных цветов, тогда как класс Color содержит некоторые предопределенные базовые цвета. Например, поле activeCaption представляет фоновый цвет строки заголовка в активном окне, поле activeCaptionText — цвет самого текста заголовка. Поле menu — цвет фона выбранного меню, поле menuText — цвет текста элемента меню, когда он не выбран, textHightlightText — цвет текста выбранного элемента меню и так далее. С помощью значения window вы можете присвоить значение цвета объекту Window, чтобы оно соответствовало другим окнам на экране пользователя, независимо от того, созданы они программой на языке Java или нет.

```
myWindow.setBackground( SystemColor.window );
```

Так как класс SystemColor является подклассом класса Color, вы можете использовать его везде, где использовали бы класс Color. Однако класс SystemColor содержит некоторые хитрости. Речь идет о константах, которые недоступны для изменения вам как программисту (ваш код не может изменить их), но они могут быть изменены системой в момент выполнения. Если пользователь изменяет свою цветовую схему, то системные цвета автоматически обновляются; в результате все, что отображено с использованием системных цветов, автоматически изменит цвет при следующей перерисовке. Например, окно myWindow должно автоматически изменить цвет фона на новый.

Класс SystemColor имеет один заметный недостаток. Вы не можете напрямую сопоставить системный цвет с Color, метод Color.equals() не

возвращает ожидаемый результат. Например, если вы хотите проверить, красный ли цвет фона, вы не можете вызывать:

```
Color.red.equals(SystemColor.window);
```

Вместо этого вы должны использовать метод `getRGB()`, чтобы найти цветовые компоненты каждого объекта и сравнить их, а не сами объекты.

Штрихование контуров формы

В то время как объекты `Graphics2D` текущего рисунка определяют, как закрашивается их форма, их текущий штрих определяет, каким будет контур формы. Для формы текущий штрих определяет такие свойства рисования, как толщина линий, пунктирные линии и стили окончаний линии.

Раньше линии могли быть только один пиксел шириной. В Java 2D в качестве толщины линий можно устанавливать значения с плавающей точкой, и в результате они могут быть объектом трансформации и масштабирования в конвейере визуализации.

Чтобы установить текущий штрих в `Graphics2D`, вызывается метод `setStroke()` в любой реализации интерфейса `Stroke`. К счастью, в 2D API включен класс `BasicStroke`, который умеет практически все. Используя `BasicStroke`, вы можете создавать пунктирные линии, выбирать украшение, добавляемое к концам линии и определять, как должны быть нарисованы углы контура.

По умолчанию `Graphics2D` использует непрерывный штрих с толщиной равной 1. В предыдущем примере Iguana толщина линии изменялась сразу перед тем, как был нарисован контур скругленного прямоугольника, вот так:

```
g2.setStroke(new BasicStroke(4));
```

Использование шрифтов

Шрифт текста в Java представлен экземпляром класса `java.awt.Font`. Объект `Font` состоит из имени, идентификатора стиля и размера пункта. Мы можем создать объект `Font` в любое время, но он значим только при применении к конкретному компоненту на данном устройстве отображения. Пример пары шрифтов:

```
Font smallFont = new Font("Monospaced", Font.PLAIN, 10);
Font bigFont = new Font("Serif", Font.BOLD, 18);
```

Имена шрифтов представлены в трех вариациях: имя семейства, физическое имя (также называемое имя шрифта) и логическое имя. Имя семейства и физическое имя могут быть тесно связаны. Например, физическое имя `Garamond Italic` соответствует семейству шрифта `Garamond`.

Логическое имя — это основное имя семейства шрифтов. Следующие логические имена шрифтов должны быть доступны на всех платформах:

- `Serif` (основное имя для `TimesRoman`)
- `SansSerif` (основное имя для `Helvetica`)
- `Monospaced` (основное имя для `Courier`)
- `Dialog`
- `DialogInput`

Логическое имя шрифта указывает на актуальный шрифт локальной платформы. Java файл `font.properties` соотносит имена шрифтов с доступными шрифтами, охватывая столько наборов символов `Unicode`, сколько возможно. Если вы запрашиваете несуществующий шрифт, вы получите шрифт, используемый по умолчанию.

Одна из крупных побед 2D API заключается в возможности использовать большинство шрифтов, установленных на вашем компьютере. Следующая программа выводит список всех шрифтов, которые доступны в 2D API:

```
//файл: ShowFonts.java
import java.awt.*;
public class ShowFonts {
   public static void main(String[] args) {
      Font[] fonts;
      fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().
getAllFonts();
      for (int i = 0; i < fonts.length; i++) {
         System.out.print(fonts[i].getFontName() + " : ");
         System.out.print(fonts[i].getFamily() + " : ");
         System.out.print(fonts[i].getName());
         System.out.println();
      }
   }
}
```

Заметим, однако, что установленные в вашей системе шрифты могут не полностью совпадать со шрифтами установленными на другой системе. Для полноценной переносимости вы можете использовать логическое имя, хотя ваше приложение и не будет выглядеть абсолютно одинаковым на всех платформах, или использовать значения по умолчанию. Как вариант, вы можете проверить существование выбранного шрифта и вернуться к логическому имени или можете позволить пользователю самому настроить шрифты приложения.

`Static` метод `Font.getFont()` ищет шрифт по имени в списке свойств системы, также как метод `Color.getColor()`. И также как метод `Color.getColor()` он интересен, но бесполезен. Обычно вы либо выбираете значение `Font` из доступных в окружении, как в примере `ShowFonts`, либо используете идентификатор для описания необходимого шрифта в конструкторе.

Класс Font определяет три идентификатора `static`: `PLAIN`, `BOLD`, и `ITALIC`. Вы можете использовать эти значения для всех шрифтов, кроме тех, которые не поддерживают стили жирный или курсив. Размер пункта определяет размер шрифта на экране. Если заданный размер не доступен, шрифт принимает размер по умолчанию.

Мы можем получить информацию о существующих шрифтах набором процедур. Методы `getName()`, `getSize()` и `getStyle()` возвращают логическое имя, размер пункта и стиль соответственно. Вы можете использовать метод `getFamily()`, чтобы найти имя семейства, а метод `getFontName()` – для получения физического имени шрифта.

Наконец, для использования объекта `Font` вы можете просто указать его в качестве аргумента метода `setFont()` объекта `Component` или `Graphics2D`. Следующая команда рисования шрифта `drawString()` использует указанный шрифт для данного компонента или в данном графическом контексте.

Метрики шрифта

Чтобы получить точную информацию о размере и значении интервала отображаемого шрифта текста, мы можем опросить объект `java.awt.font.LineMetrics`. В разных системах доступны разные шрифты, и среди них может и не быть шрифта, который вы запрашиваете. Кроме того, размеры различных символов внутри одного шрифта могут быть разными, особенно в многоязычных текстах. Так объект `LineMetrics` пред-

ставляет информацию о конкретном наборе текста конкретного шрифта в конкретной системе, а не общую информацию о шрифте. Например, если вы запрашиваете метрику девятипунктного шрифта `Monospaced`, то получите не абстрактную информацию о шрифте `Monospaced`, а метрики шрифта, который конкретная система использует как девятипунктный шрифт `Monospaced` и который может и не быть ровно девятипунктным или даже моноширинным.

Используйте метод `getLineMonospaced()` для объекта `Font`, чтобы получить метрики для текста в том виде, в котором он должен появиться в данном компоненте. Этому методу также нужно знать некоторую информацию о том, как вы собираетесь визуализировать текст, если вы планируете использовать сглаживание для экземпляра, которое влияет на измерения текста. Дополнительная информация инкапсулирована в классе `FontRenderContext`. К счастью, вы можете просто запросить у `Graphics2D` его `FontRenderContext` вместо создания собственного:

```
public void paint(Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   ...
   FontRenderContext frc = g2.getFontRenderContext();
   LineMetrics metrics = font.getLineMetrics("Monkey", frc);
   ...
}
```

Класс `Font` также содержит метод `getStringBounds()`, который возвращает границы пространства участка текста:

```
public void paint(Graphics g) {
   Graphics2D g2 = (Graphics2D)g;
   ...
   FontRenderContext frc = g2.getFontRenderContext();
   float messageWidth =
       (float)font.getStringBounds("Monkey", frc).getWidth();
   ...
}
```

Следующее приложение `FontShow` отображает слово и рисует опорные линии, показывая определенные характеристики шрифта, как показано на рис 20.3. Нажимая мышью в окне приложения, переключите размер пункта между мелким и крупным.

```
//файл: FontShow.java
import java.awt.*;
import java.awt.event.*;
import java.awt.font.*;
import javax.swing.*;
public class FontShow extends JComponent
{
   private static final int PAD = 25; // линия отступа
   private boolean bigFont = true;
   private String message;
   public FontShow(String message) {
      this.message = message;
   addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
         bigFont = !bigFont;
         repaint();
      }
   });
}
public void paint(Graphics g)
{
   Graphics2D g2 = (Graphics2D)g;
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);
   int size = bigFont ? 96 : 64;
   Font font = new Font("Dialog", Font.PLAIN, size);
   g2.setFont(font);
   int width = getSize().width;
   int height = getSize().height;
   FontRenderContext frc = g2.getFontRenderContext();
   LineMetrics metrics = font.getLineMetrics(message,          frc);
   float messageWidth =
    (float)font.getStringBounds(message, frc).getWidth();
   // текст в центре
   float ascent = metrics.getAscent();
   float descent = metrics.getDescent();
   float x = (width - messageWidth) / 2;
   float y = (height + metrics.getHeight()) / 2 - descent;
   g2.setPaint(getBackground());
   g2.fillRect(0, 0, width, height);
   g2.setPaint(getForeground());
```

```
        g2.drawString(message, x, y);
        g2.setPaint(Color.white); // линии шрифта
        drawLine(g2, x - PAD, y, x + messageWidth + PAD, y);
        drawLine(g2, x, y + PAD, x, y - ascent - PAD);
        g2.setPaint(Color.green); // верхняя линия
        drawLine(g2, x - PAD, y - ascent,
        x + messageWidth + PAD, y - ascent);
        g2.setPaint(Color.red); // нижняя линия
        drawLine(g2, x - PAD, y + descent,
        x + messageWidth + PAD, y + descent);
    }
    private void drawLine(Graphics2D g2,
        double x0, double y0, double x1, double y1) {
        Shape line = new java.awt.geom.Line2D.Double(x0, y0,              x1,
y1);
        g2.draw(line);
    }
    public static void main(String args[]) {
        String message = "Lemming";
        if (args.length > 0) message = args[0];
        JFrame frame = new JFrame("FontShow");
        frame.setSize(420, 300);
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.getContentPane().add(new FontShow(message));
        frame.setVisible(true);
    }
}
```

Вы можете указать текст, который будет отображен, как аргумент командной строки:

```
% java FontShow "When in the course of human events ..."
```

Приложение FontShow может выглядеть немного сложным, но на самом деле это не так. Большая часть кода — это метод `paint()`, который устанавливает шрифт, рисует текст и добавляет несколько линий в качестве иллюстрации некоторых характеристик (метрик) текста. Для забавы мы отслеживаем нажатие мыши (используя обработчик событий определенный в конструкторе) и изменяем размер шрифта, изменяя переменную-переключатель `bigFont` и перерисовывая.

По умолчанию, текст визуализируется выше и справа относительно координат, установленных методом `drawString()`. Воспринимайте эту

начальную точку как начало координатной системы; оси — это линии шрифта.

Приложение FontShow рисует эти линии белым цветом. Наибольшая высота символов над линией шрифта называется *подъем* и показана зеленой линией. В некоторых шрифтах часть символов опущена ниже линии шрифта.

Наибольшее расстояние, на которое элемент символа выступает ниже линии шрифта, называется *спуск*, он отображен красной линией.

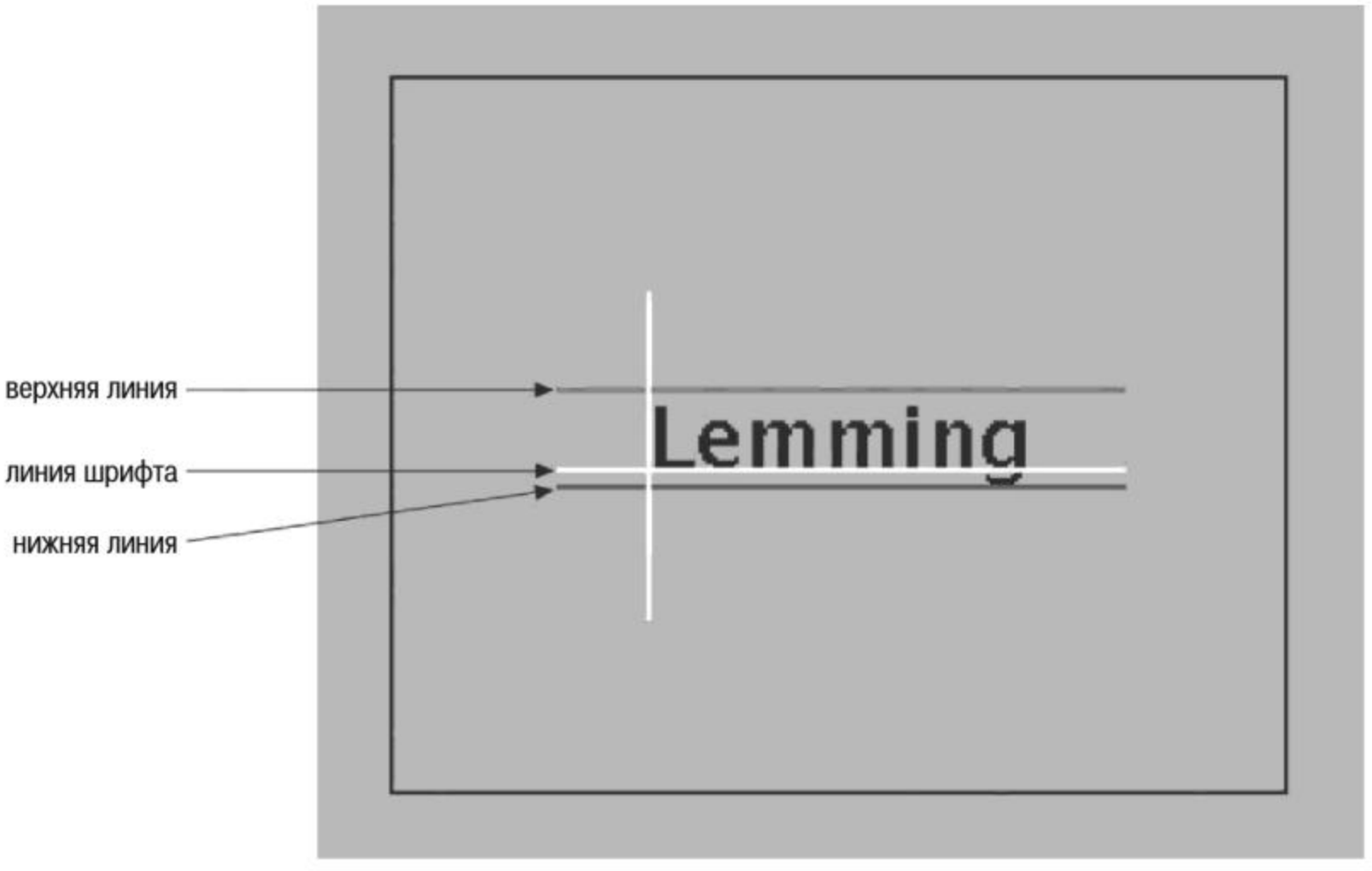

Рис. 20.3. Приложение FontShow

Мы узнаем о подъеме и спуске нашего шрифта, используя методы `getAscent()` и `getDescent()` класса `LineMetrics`. Также мы запрашиваем ширину нашей строки (когда она визуализирована данным шрифтом) используя метод `getStringBounds()` класса `Font`.

Эта информация используется для размещения слова в центре области отображения. Для вертикального центрирования слова мы используем высоту и сверяем со спуском, чтобы вычислить размещение линии шрифта.

В табл. 20.2 представлен краткий список методов, которые возвращают полезные метрики шрифта.

Табл. 20.2. Методы класса `LineMetrics`

Метод	Описание
`getAscent()`	Высота над линией шрифта
`getDescent()`	Глубина под линией шрифта
`getLeading()`	Стандартное расстояние между строками
`getHeight()`	Итоговая высота строки (подъем+спуск+межстрочный интервал)

Межстрочный интервал — это отступ между строками текста. Метод `getHeight()` сообщает итоговую высоту строки текста, включая расстояние между строками

Вывод изображений

До этого момента мы работали с методами, рисующими простые формы и отображающими текст. Чтобы использовать более сложную графику, мы будем работать с изображениями. В стандартном Swing приложении простейший способ отобразить изображение в вашем приложении — использовать класс `ImageIcon` и компонент `JLabel`. Здесь мы говорим о работе с обрабатываемым изображением на низком уровне: рисовании. 2D API содержит мощный набор инструментов для создания и отображения обрабатываемого изображения. Мы начнем с основного класса `java.awt.Image` и увидим, как загрузить изображение в приложение, и нарисовать его там, где вы хотите. Инструментарий AWT будет обрабатывать большинство деталей автоматически. В следующей главе мы также продолжим изучать ручное управление загрузкой изображения также как и создание и управление необработанными пиксельными данными, позволяющими вам создавать любой вид графики, о каком бы вы ни мечтали.

Ядро AWT поддерживает изображения форматов JPEG, PNG, и GIF. Сюда также включен формат анимации GIF89a так, что вы можете работать с простой анимацией так же легко, как со статическими изображениями. Если вам необходимо работать с другими типами изображений, вы можете подключить фреймворк Java Advanced Imaging `javax.imageio`. Мы кратко остановимся на нем здесь и в следующей главе, когда будем обсуждать класс `BufferedImage`.

В большинстве случаев фреймворк `ImageIO` заменяет и замещает старую функциональность обработки изображений ядра AWT так же,

как Swing расширяет и замещает старые компоненты AWT. Фреймворк `ImageIO` легко расширяется для поддержки новых типов изображений через плагины. Однако все, что он добавляет в терминах поддержки типов изображений — это возможность чтения растрового (BMP) и оптимизированного для мобильных устройств растрового (WBMP) формата изображений. Поскольку в большей части кода, написанного на Java, используется оригинальная функциональность AWT, мы сосредоточимся именно на ней.

Класс `Image`

Класс `java.awt.Image` представляет вид изображения. Вид создается из изображения источника, которое предоставляет обрабатываемые пикселы. Изображение может быть из статического источника, как файл JPEG, или динамического, как видеопоток или графический процессор.

AWT `Images` создаются методами `getImage()` и `createImage()` класса `java.awt.Toolkit`. Существует две формы каждого метода, которые принимают URL или простой путь к файлу соответственно.

Метод `createImage()` может также прямо получать байтовый массив обрабатываемого изображения.

Когда вы создаете изображение в своем приложении, вы должны использовать метод `getResource()` класса `Class` (обсуждалось в главе 1) для составления URL ссылающегося на файл из директории приложения. Метод `getResource()` позволяет вам связывать изображение с вашим приложением, внутри JAR-файлов или где-либо еще в директории `classpath`. Следующий фрагмент кода показывает некоторые примеры загрузки изображения методом `getImage()`:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
// URL ресурсов приложения -лучший метод
URL daffyURL = getClass().getResource("/cartoons/images/daffy.gif");
Image daffyDuckImage = toolkit.getImage( daffyURL );
// Абсолютный URL -
URL monaURL = new URL( "http://myserver/images/mona_lisa.png");
Image monaImage = toolkit.getImage( monaURL );
// Локальный файл -
Image elvisImage = toolkit.getImage("c:/elvis/lateryears/fatelvis1.jpg" );
```

Метод `createImage()` выглядит точно так же, как метод `getImage()`; разница в том, что метод `getImage()` "затормаживает" изображения и разделяет их при получении множественных запросов одинаковых данных. Метод `createImage()` не делает этого, он каждый раз создает новый объект Image и полагается на вас в части кеширования и разделения изображений. Метод `getImage()` удобно применять в приложениях, использующих ограниченное количество изображений за время своей работы, но он может выводить данные изображений. Вы должны использовать метод `createImage()` и кешировать объект `Image` самостоятельно.

Класс `javax.imageio.ImageIO` похожим образом предоставляет несколько статических методов `read()`, которые могут загружать изображения из следующих источников: `File`, `URL`, или `InputStream`.

```
URL daffyURL = getClass().getResource("/cartoons/images/daffy.gif");
Image daffyDuckImage = ImageIO.read( daffyURL );
```

Мы обсудим загрузку изображений с использованием AWT и фреймворка ImageIO более подробно в главе 21.

Когда у нас есть объект `Image`, мы можем нарисовать его в графическом контексте, используя метод `drawImage()` класса `Graphics`. Простейшая форма метода `drawImage()` требует четырех параметров: объект `Image`, координаты `x`, `y` в которых он будет нарисован, и ссылка на специальный объект обозреватель изображения. Скоро мы покажем пример вызова метода `drawImage()`, но сперва нужно сказать несколько слов об обозревателе изображений.

Обозреватели изображений

Изображения обрабатываются асинхронно — это означает, что Java выполняет операции над изображениями, такие как загрузка и масштабирование, в фоновом режиме, позволяя продолжать выполнение пользовательского кода. В стандартном клиентском приложении это зачастую не важно, изображения могут быть размером с кнопки и, как правило, они связаны с приложением для почти мгновенного поиска. Однако платформа Java работает изображениями через веб так же хорошо, как и локально, и вы увидите, как это проявляется в API для работы с изображениями.

Например, метод `getImage()` всегда возвращается незамедлительно, даже если обрабатываемое изображение было отправлено вам через сеть

с Марса и еще не доступно. Фактически, если это новое изображение, Java даже не начнет показывать данные, пока мы не пытаемся отобразить его или управлять им. Достоинство этой техники в том, что Java может работать с мощной, многозадачной средой обработки изображений автоматически. Однако это также привносит несколько проблем. Java загружает изображение автоматически, как мы узнаем, когда оно полностью загружено? Что если мы хотим работать с изображением, когда оно поступило? Что если нам необходимо знать свойства изображения (например, его размеры) до начала работы? Что если произошла ошибка во время загрузки изображения? С этими проблемами призваны справиться обозреватели изображений, объекты, реализующие интерфейс `ImageObserver`. Все операции, которые рисуют или проверяют объекты класса `Image`, асинхронны и принимают обозреватель изображений в качестве параметра. `ImageObserver` отслеживает статус операций над изображением и может сделать эту информацию доступной остальной части приложения. Когда обрабатываемое приложение загружается графической системой, ваш обозреватель изображения уведомляет о его прогрессе, включая моменты, когда доступны новые пикселы, когда готов полный кадр изображения, и если произошла ошибка при загрузке. Обозреватель изображения также получает, как атрибут, информацию об изображении — такую как размеры и свойства — как только она становится известна.

Метод `drawImage()`, как и другие операции над изображением, ссылается на объект `ImageObserver`, как на параметр. Метод `drawImage()` возвращает значение типа `Boolean`, определяющее, было ли изображение нарисовано полностью. Если обрабатываемое изображение еще не было загружено или доступно только частично, метод `drawImage()` рисует ту часть изображения, которую может, и возвращает. В фоновом режиме графическая система начинает (или продолжает) загрузку обрабатываемого изображения. Объект обозреватель изображения регистрируется как заинтересованный в информации об изображении. Далее обозреватель повторно вызывается при поступлении дополнительной информации о доступных пикселах и еще раз, когда изображение будет загружено целиком. Обозреватель изображения может делать с этой информацией все что угодно. Наиболее часто информация используется при вызове метода `repaint()`, чтобы приложение заново нарисовало изображение с обновленными данными. В этом случае приложение или апплет могут рисовать изображение по мере его поступления, создавая эффект постепенной загрузки. Или же они могут ждать загрузки всего изображения, прежде чем его отобразить. Обозреватели изображения

освещены в главе 21. Сейчас давайте опустим вопрос использования готового обозревателя изображения. Класс `Component` реализует интерфейс `ImageObserver` и предоставляет некоторые простые способы рисования — это означает, что каждый компонент может обрабатываться в качестве собственного обозревателя изображения по умолчанию. Мы можем просто передать ссылку на какой либо компонент, выполняющий рисование, в качестве параметра обозревателя изображения при вызове метода `drawImage()`:

```
public void paint( Graphics g ) {
g.drawImage( monaImage, x, y, this );
...
```

Наш компонент обрабатывается как обозреватель изображения и вызывает метод `repaint()` самостоятельно, чтобы перерисовать изображение при необходимости. Если изображение поступает медленно, наш компонент уведомляется каждый раз, когда становится доступным новый фрагмент. В результате изображение появляется постепенно, в процессе загрузки*.

Предварительная загрузка изображений

Мы обсудим загрузку изображения более детально в следующей главе, когда будем рассматривать утилиту MediaTracker, которая следит за процессом загрузки одного или более изображений. Однако мы заглянем вперед прямо сейчас и покажем вам простой кратчайший способ загрузки одиночного изображения и проверки, что загрузка завершена и готова к рисованию. Вы можете использовать класс `javax.swing.ImageIcon` для автоматического выполнения грязной работы:

```
ImageIcon icon = new ImageIcon("myimage.jpg");
Image image = icon.getImage();
```

Изображения, загруженные методом `read()` класса `ImageIO`, возвращаются полностью загруженными. Класс `ImageIO` предоставляет собственный API для слежения за процессом загрузки изображения. Этот API следует более стандартным шаблонам событий источников/слушателей, но мы не хотим использовать их здесь.

---

* Системные свойства `awt.image.incrementaldraw` и `awt.image.redrawrate` контролируют это поведение. Свойство `redrawrate` ограничивает частоту вызова метода `repaint()`; значение по умолчанию равно 100 миллисекундам. Значение по умолчанию свойства `incrementaldraw` — `true`, задействует это поведение. Установив его равным `false`, вы устанавливаете задержку рисования до поступления всего изображения.

Размер и масштабирование

Другая версия метода `drawImage()` визуализирует масштабированную версию изображения:

```
g.drawImage( monaImage, x, y, x2, y2, this );
```

Так рисуется все изображение внутри прямоугольника, образованного точками `x`, `y` и `x2`, `y2`, при необходимости оно масштабируется. Метод `drawImage()` поступает почти так же, как раньше: изображение обрабатывается компонентом при его поступлении, обозреватель изображения уведомляется при поступлении новых обрабатываемых пикселов и когда все изображение полностью доступно. Несколько других перегруженных версий метода `drawImage()` предоставляют более сложные функции: вы можете масштабировать, обрезать изображение и выполнять некоторые простые операции трансформации.

Однако обычно из соображений производительности вы хотите сделать увеличенную копию изображения (вместо того чтобы каждый раз отрисовывать ее). Для этой цели вы можете использовать метод `getScaledInstance()`. Таким образом:

```
Image scaledDaffy =
daffyImage.getScaledInstance( 100, 200, Image.SCALE_AREA_AVERAGING );
```

Этот метод увеличивает оригинальное изображение до заданного размера, 100 в 200 пикселов. Он возвращает новый объект `Image`, который вы рисуете, как любое другое изображение. Константа `SCALE_AREA_AVERAGING` сообщает методу `getScaledImage()` какой алгоритм масштабирования использовать. Алгоритм, используемый здесь, пытается выполнить скромную работу увеличения, расходуя время. Есть некоторые варианты, которые требуют меньше времени: `SCALE_REPLICATE` увеличивает изображение повторяя его строки и столбцы, такой алгоритм быстрее, но дает менее хороший результат. Также вы можете указать либо `SCALE_FAST`, либо `SCALE_SMOOTH` и позволить реализации выбрать соответствующий алгоритм, оптимизирующий временные затраты или качество. Если у вас нет определенных требований, вы должны использовать константу `SCALE_DEFAULT`, которая в идеале установит предпочтения пользовательской среды.

Если вы собираетесь нарисовать изображение более одного раза, как обычно и бывает, создание увеличенной копии изображения позволяет существенно повысить производительность. Другими словами, повтор-

ный вызов метода `drawImage()` с требованием масштабирования каждый раз вызывает масштабирование изображения.

Методы класса `Image` `getHeight()` и `getWidth()` возвращают размеры изображения. Поскольку эта информация недоступна до окончательной загрузки изображения, оба метода также получают объект `ImageObserver` в качестве параметра. Если измерения еще не доступны, они возвращают значение –1 и уведомляют обозреватель, когда актуальные значения известны. Мы увидим, как разобраться с этой и другими проблемами немного позже. Пока мы продолжим использовать наш объект `Component`, как обозреватель изображения и перейдем к некоторым основным техникам рисования.

Техники рисования

Теперь, когда мы знаем основные инструменты, попробуем совместить некоторые из них. В этом разделе мы увидим некоторые техники, которые быстро работают и рисуют без мерцания. Если вас интересует анимация, то это как раз для вас. Операции рисования требуют времени, и время, затрачиваемое на рисование, приводит к задержкам и снижению качества результатов. Наша цель минимизировать количество выполняемых нами действий рисования, насколько это возможно, выполняя работу в стороне от глаз пользователя. Для этого мы используем две техники: отсечение и двойная буферизация. К счастью, Swing сейчас работает с двойной буферизацией по умолчанию. Вам не нужно реализовывать эту логику самостоятельно, но полезно понимать ее. Наш первый пример DragImage иллюстрирует некоторые вопросы обновления экрана. Он состоит из двух частей, что типично для анимации: постоянный фон и изменяемый объект на переднем плане. В этом случае фон — это шахматная доска, а объект — небольшое, масштабируемое изображение, которое мы можем перемещать по ней, как показано на рис. 20.4:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DragImage extends JComponent
implements MouseMotionListener
{
static int imageWidth=60, imageHeight=60;
int grid = 10;
```

```
int imageX, imageY;
Image image;
public DragImage(Image i) {
image = i;
addMouseMotionListener(this);
}
public void mouseDragged(MouseEvent e) {
imageX = e.getX();
imageY = e.getY();
repaint();
}
public void mouseMoved(MouseEvent e) {}
public void paint(Graphics g) {
Graphics2D g2 = (Graphics2D)g;
int w = getSize().width / grid;
int h = getSize().height / grid;
boolean black = false;
for (int y = 0; y <= grid; y++)
for (int x = 0; x <= grid; x++) {
g2.setPaint(black ? Color.black : Color.white);
black = !black;
g2.fillRect(x * w, y * h, w, h);
}
g2.drawImage(image, imageX, imageY, this);
}
public static void main(String[] args) {
String imageFile = "L1-Light.jpg";
if (args.length > 0)
imageFile = args[0];
//Отключение двойной буферизации
//RepaintManager.currentManager(null).setDoubleBufferingEnabled(false);
Image image = Toolkit.getDefaultToolkit().getImage(
DragImage.class.getResource(imageFile));
image = image.getScaledInstance(
imageWidth,imageHeight,Image.SCALE_DEFAULT);
JFrame frame = new JFrame("DragImage");
frame.add( new DragImage(image) );
frame.setSize(300, 300);
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setVisible(true);
}
}
```



Рис. 20.4. Приложение DragImage

Запустите приложение, дополнительно указав файл изображения, как аргумент командной строки. Затем попробуйте переместить изображение по доске.

`DragImage` — это стандартный компонент, переопределяющий метод `paint()` класса `JComponent` для выполнения рисования. В методе `main()` мы загружаем изображение и предварительно масштабируем для улучшения качества. Далее мы создаем компонент `DragImage` и размещаем его в панели содержимого. Когда мышь двигается, `DragImage` отслеживает ее текущее положение в двух переменных экземпляра `imageX` и `imageY`. При каждом вызове метода `mouseDragged()`, координаты обновляются, и вызывается метод `repaint()`, чтобы обновить экран. Когда метод `paint()` вызывается, он ищет некоторые параметры, рисует шахматную доску для заполнения области апплета и в завершение рисует небольшую версию изображения в последних координатах.

Теперь несколько тайных подробностей о различиях между `JComponent` и обычным AWT `Component`. Сначала, стандартный метод `update()` класса `JComponent` просто вызывает наш метод `paint()`. До Java 1.4, стандартный метод `update()` класса AWT `Component` предварительно перед вызовом `paint` очищал область экрана, используя метод `clearRect()`. Напомним, что различие между методами `paint()` и `update()` состоит в том, что метод `paint()` рисует всю область, а метод `update()` предполагает, что участок экрана остался неизменным с предыдущего рисования. В AWT метод `update()` крайне консервативен; в Swing он более оптимистичен. Это требует внимания, если вы работаете со старым AWT приложением. В таком случае, вы можете переопределить метод `update()` для вызова метода `paint()`.

Более важное различие между AWT и Swing в том, что Swing компоненты по умолчанию используют двойную буферизацию при выводе их метода `paint()`.

Двойная буферизация

Двойная буферизация означает, что вместо рисования непосредственно на экране, сначала Swing выполняет операции рисования во внеэкранном буфере, затем копирует завершенную работу на экран за одну операцию рисования, как показано на рис. 20.5. Необходимо некоторое время для выполнения работы рисования, но когда она выполнена, двойная буферизация мгновенно обновляет наш экран так, что пользователь не заметит никакого мерцания или постепенного вывода визуализации.

Вы увидите, как реализовать эту технику самостоятельно, когда мы будем использовать внеэкранный буфер, позже в этой главе. Однако Swing выполняет этот вид двойной буферизации всегда, когда вы используете компонент Swing в контейнере Swing. Компоненты AWT не имеют возможности автоматической двойной буферизации.

Ради интереса возьмем наш пример и отключим двойную буферизацию, чтобы увидеть эффект. Каждый объект Swing `JComponent` содержит метод, называемый `setDoubleBuffered()`, который может быть установлен равным false для отключения техники. Или вы можете отключить ее для всех компонентов, воспользовавшись вызовом Swing `RepaintManager`, как мы отметили в комментарии в примере. Попробуйте раскомментировать эту строку приложения DragImage, и отследить проявившиеся различия.

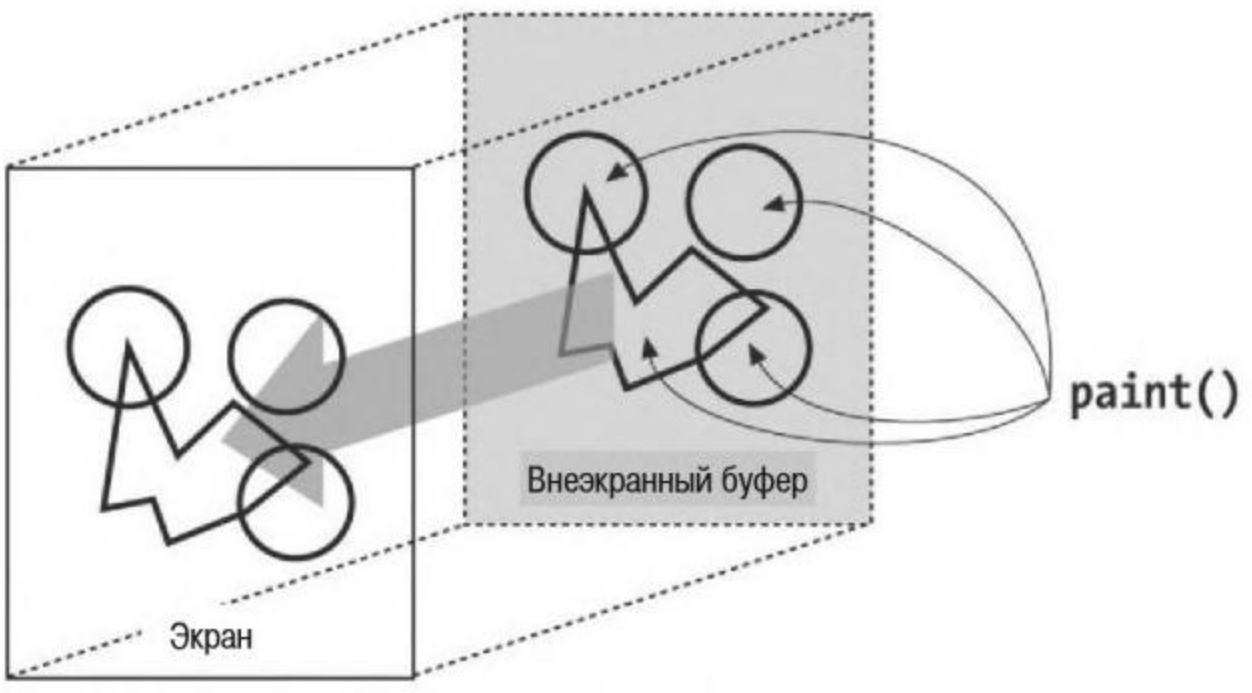

Рис. 20.5. Двойная буферизация

Разница наиболее заметна, когда вы используете медленную систему или выполняете сложные операции рисования. Двойная буферизация

устраняет любое мерцание. Однако на медленной системе это может заметно ухудшить качество. В крайних случаях, например, в играх, может быть полезной возможность отключать двойную буферизацию.

Наш пример крайне быстр, но мы все еще выполняем впустую часть рисования. Большая часть фона каждый раз остается неизменной. Вы могли бы попытаться сделать метод `paint()` умнее так, что он не будет перерисовывать эту область, но напомним, что метод `paint()` обязан уметь рисовать сцену целиком потому, что он может быть вызван в ситуации, когда экран изменен. Решение: рисовать только часть картинки при любом движении мыши. Далее мы поговорим про отсечение.

Ограниченное рисование с отсечением

В любой момент, когда происходит движение мыши, приложение DragImage отвечает обновлением координат и вызовом метода `repaint()`. Но метод `repaint()` по умолчанию вызывает перерисовывание всего компонента. В большей части этой работы нет необходимости. Оказывается, существует другая версия метода `repaint()`, позволяющая вам определять прямоугольную область, которая должна быть перерисована, в сущности, область отсечения.

Как может помочь ограничение области рисования? Прежде всего, операции рисования, которые выходят за пределы региона отсечения, не отображаются. Если операция рисования перекрывает область отсечения, мы увидим только ту часть, которая находится внутри. Второй эффект в том, что при правильной реализации графический контекст может распознавать операции рисования, полностью выходящие за пределы региона отсечения и игнорировать их все. Уничтожение ненужных операций может сохранить время, если мы делаем что-то сложное, например закрашиваем кучу многоугольников. Это не сэкономит времени, которое наше приложение тратит на отправку вызова методов рисования, но затраты на вызов таких методов обычно ничтожны в сравнении со временем, затрачиваемым на их выполнение. Если мы создаем изображение пиксел за пикселом, это другой случай, так как основное время поглощают вычисления, а не рисование.

Таким образом, мы можем сэкономить некоторое время нашего приложения, прерисовывая только нужные части экрана. Мы можем указать мельчайшую прямоугольную область, которая включает и старое, и новое положение изображения, как показано на рис. 20.6. Это един-

ственный участок экрана, который действительно необходимо изменить, все остальное остается неизменным.

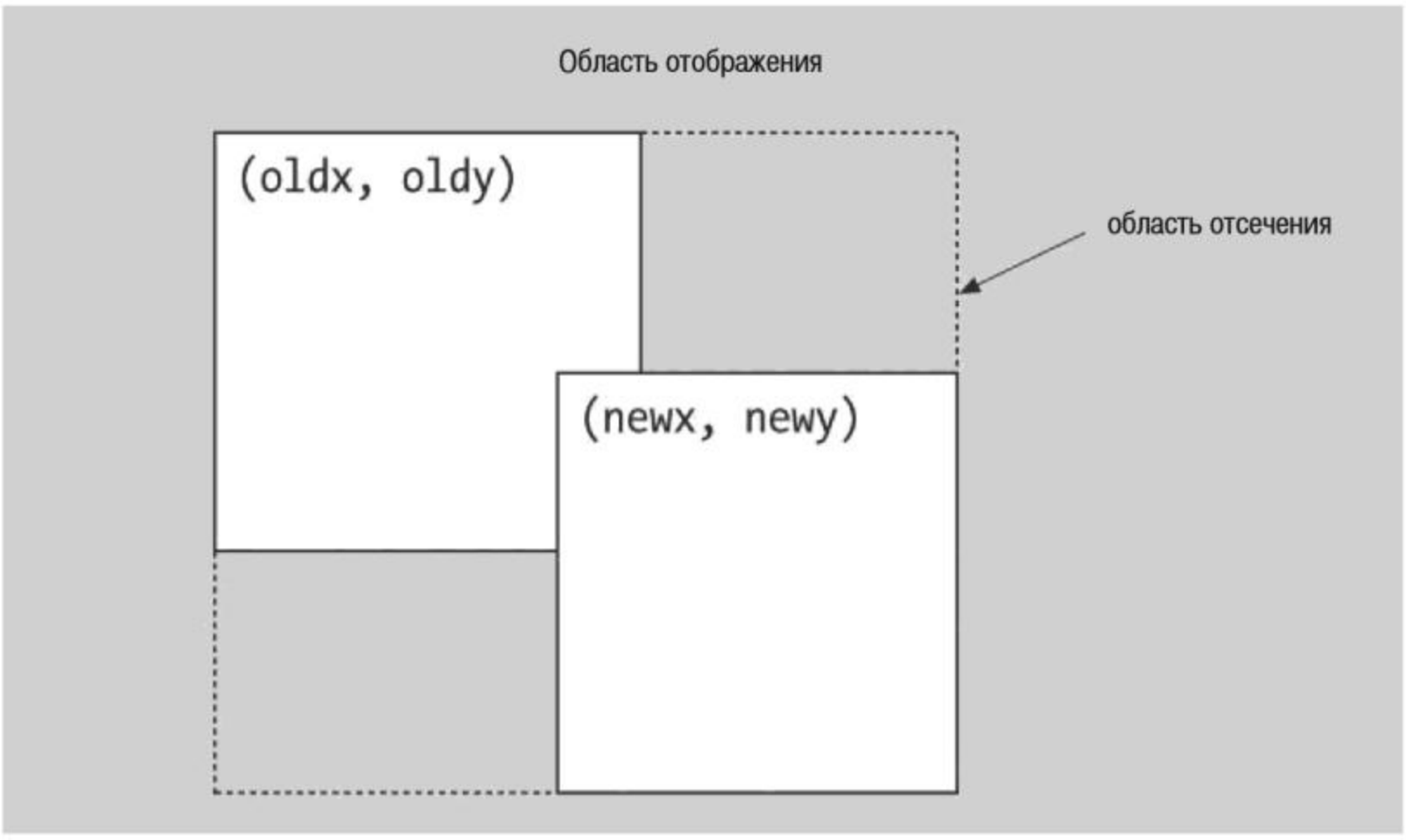

Рис. 20.6. Определение области отсечения

Более умный алгоритм может экономить даже больше времени, перерисовывая только ту область, которая была изменена. Однако реализованная нами простая стратегия отсечения может быть применена ко множеству видов рисования и быть достаточно производительной, особенно если измененная область мала. Обратите внимание на одну важную вещь: в дополнение к поиску новой позиции, наша операция обновления теперь запоминает последнюю позицию, в которой изображение было нарисовано. Давайте исправим наше приложение так, чтобы оно использовало определенную область отсечения. Чтобы сохранить его краткость и подчеркнуть изменения, мы создадим наш следующий пример подкласса `DragImage`. Назовем его `ClippedDragImage`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ClippedDragImage extends DragImage {
int oldX, oldY;
public ClippedDragImage( Image i ) { super(i); }
public void mouseDragged(MouseEvent e) {
imageX = e.getX();
```

```
imageY = e.getY();
Rectangle r = getAffectedArea(
oldX, oldY, imageX, imageY, imageWidth, imageHeight);
repaint(r); //перерисовка только необходимую часть компонента
oldX = imageX;
oldY = imageY;
}
private Rectangle getAffectedArea(
int oldx, int oldy, int newx, int newy, int width, int height)
{
int x = Math.min(oldx, newx);
int y = Math.min(oldy, newy);
int w = (Math.max(oldx, newx) + width) - x;
int h = (Math.max(oldy, newy) + height) - y;
return new Rectangle(x, y, w, h);
}
public static void main(String[] args) {
String imageFile = "L1-Light.jpg";
if (args.length > 0)
imageFile = args[0];
//Отключение двойной буферизации
//RepaintManager.currentManager(null).setDoubleBufferingEnabled(false);
Image image = Toolkit.getDefaultToolkit().getImage(
ClippedDragImage.class.getResource(imageFile));
image = image.getScaledInstance(
imageWidth,imageHeight,Image.SCALE_DEFAULT);
JFrame frame = new JFrame("ClippedDragImage");
frame.add( new ClippedDragImage(image) );
frame.setSize(300, 300);
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setVisible(true);
}
}
```

Вы можете заметить (а можете и не заметить), что `ClippedDragImage` значительно быстрее. Современные настольные компьютеры настолько быстры, что подобные операции для них — детская игра. Однако эта фундаментальная техника важна и применима в более сложных приложениях.

Что мы изменили? Во-первых, мы переопределили метод `mouseDragged()` таким образом, что вместо установки текущих координат изображения он выясняет область, которая изменена применением ново-

го метода `private`. Метод `getAffectedArea()` получает новые и старые координаты, ширину и высоту изображения в качестве аргументов. Он определяет ограничивающий прямоугольник, как показано на рис. 20.6, далее вызывается метод `repaint()` для рисования только необходимой части экрана. Метод `mouseDragged()` также сохраняет текущее положение, устанавливая переменные `oldX` и `oldY`.

Попробуйте отключить двойную буферизацию в этом примере и сравните с предыдущим примером без использования буферизации, чтобы увидеть, как мало работы было сделано. Возможно, вы не увидите разницы, компьютеры сегодня достаточно быстры. Если бы вы использовали 2D API для выполнения какой-то замысловатой визуализации, это бы очень помогло.

Внеэкранное рисование

В дополнение к подготовке буфера двойной буферизации, внеэкранное изображение полезно для сохранения сложной фоновой информации, тяжелой в реализации. Мы рассмотрим забавный простой пример блокнота для каракулей. DoodlePad — это простой инструмент рисования, позволяющий нам создавать каракули движением мыши, как показано на рис. 20.7. Рисование происходит во внеэкранном изображении, метод `paint()` просто копирует изображение на экран:

```
//файл: DoodlePad.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DoodlePad
{
public static void main(String[] args)
{
JFrame frame = new JFrame("DoodlePad");
frame.setLayout(new BorderLayout());
final DrawPad drawPad = new DrawPad();
frame.add(drawPad, BorderLayout.CENTER);
JPanel panel = new JPanel();
JButton clearButton = new JButton("Clear");
clearButton.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
drawPad.clear();
```

```
}
});
panel.add(clearButton);
frame.add(panel, BorderLayout.SOUTH);
frame.setSize(280, 300);
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setVisible(true);
}
} // Конец класса DoodlePad
class DrawPad extends JComponent
{
Image image;
Graphics2D graphics2D;
int currentX, currentY, oldX, oldY;
public DrawPad() {
setDoubleBuffered(false);
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent e) {
oldX = e.getX();
oldY = e.getY();
}
});
addMouseMotionListener(new MouseMotionAdapter() {
public void mouseDragged(MouseEvent e) {
currentX = e.getX();
currentY = e.getY();
if (graphics2D != null)
graphics2D.drawLine(oldX, oldY, currentX, currentY);
repaint();
oldX = currentX;
oldY = currentY;
}
});
}
public void paintComponent(Graphics g) {
if (image == null) {
image = createImage(getSize().width, getSize().height);
graphics2D = (Graphics2D)image.getGraphics();
graphics2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
clear();
```

```
}
g.drawImage(image, 0, 0, null);
}
public void clear() {
graphics2D.setPaint(Color.white);
graphics2D.fillRect(0, 0, getSize().width, getSize().height);
graphics2D.setPaint(Color.black);
repaint();
}
}
```



Рис. 20.7. Приложение DoodlePad

Опробуйте наше приложение. Нарисуйте симпатичного лося или закат. Мы нарисовали замечательный мультфильм о Билле Гейтсе. Если вы допустили ошибку, нажмите кнопку Clear и начните сначала.

Сейчас все части должны быть вам знакомы. Мы создали тип `JComponent` и назвали его DrawPad.

Новый компонент DrawPad использует внутренние классы для применения обработчиков интерфейсов `MouseListener` и `MouseMotionListener`. Мы использовали JComponent, создав метод `Image()`, чтобы создать пустой внеэкранный буфер изображения для хранения наших каракулей. Событие перемещения мыши (Mouse-dragging) побуждает нас рисовать линии во внеэкранном изображении и вызвать метод `repaint()`, для обновления экрана. Метод `paint()` приложения DrawPad создает `drawImage()` для копирования внеэкранной области рисования на экран. В этом случае DrawPad сохраняет информацию нашего наброска.

Необычно в DrawPad то, что он частично выполняет рисование вне метода `paint()`. В этом примере мы хотим позволить пользователю рисовать каракули мышью и отвечаем на каждое движение мыши. Поэтому мы делаем нашу работу, рисуем во внеэкранном буфере методом

Dragged(). Как правило, мы должны с осторожностью выполнять тяжелую работу в методах обработки событий, потому что мы не хотим пересекаться с другими заданиями, которые представляют процессы рисования в оконной системе. В этом случае наша функция рисования линии не должна быть нагрузкой и главное наше беспокойство должна вызвать возможность связать события перемещения мыши и вывод на экран. Более сложный пример должен передавать координаты в очередь, для других потоков рисования тем самым освобождая поток обработчика событий.

В дополнение к рисованию линии, когда пользователь перемещает мышь, обработчик mouseDragged() содержит пару предыдущих координат, чтобы использовать их как начальную точку для следующего участка линии. Обработчик mousePressed() переустанавливает предыдущие координаты на текущую позицию мыши каждый раз, когда пользователь перемещает мышь. В завершение DrawPad предоставляет метод clear(), который очищает внеэкранный буфер и вызывает метод repaint() для обновления экрана. Приложение DoodlePad вызывает метод clear()для кнопки с соответствующим именем через другой анонимный внутренний класс.

Что если мы хотим сделать что-то с изображением после того, как пользователь закончил рисовать? Как мы увидим в следующей главе, мы можем получить информацию о пикселах и работать с ней. Создать объект, хранящий обрабатываемые пикселы, чтобы позже воспроизвести их, будет несложно. Подумайте, как можно было бы создать сетевое приложение , в котором люди могли бы рисовать каракули на вашей веб-странице.

Печать

Ранее в этой главе мы намекали на то, что у вас есть возможность делать изображения одинаковыми на экране и на печати. Это действительно так, все, что вам необходимо сделать — получить объект Graphics2D, который определяет, куда осуществлять вывод: на принтер или в область экрана. Java Printing API предоставляет необходимые инструменты. Нам не хватит места в этой главе, чтобы описать Printing API полностью, но на коротком примере мы покажем то, что заинтересует вас.

Классы печати собраны в пакете java.awt.print. Вы можете напечатать все что угодно, если это реализует интерфейс Printable. Как вы

уже догадались, этот интерфейс содержит только один метод `print()`, который аналогичен методам `paint()`, с которыми мы уже работали. Метод получает доступ к объектам `Graphics`, которые представляют нарисованную поверхность на странице печати. Он также получает доступ к объекту `PageFormat`, который инкапсулирует информацию о формате бумаги, на которой вы печатаете. В завершение метод `print()` получает число страниц перед началом визуализации. Все компоненты библиотеки Swing реализуют метод `print()`, который вы можете использовать или переписать для настройки их печатного представления.

Ваша реализация метода `print()` должна либо выполнить визуализацию указанной страницы, либо сообщить, что ее не существует. Вы можете выполнить это, возвращая специальные значения из метода `print()`: `Printable.PAGE_EXISTS`, либо `Printable.NO_SUCH_PAGE`.

Вы можете контролировать задание печати, в том числе появление диалогового окна настроек печати, используя класс `PrinterJob`. Следующий класс даст вам возможность поместить что-либо на бумагу. В этом примере мы работаем с обеими частями уравнения печати: реализуем простой интерфейс `Printable` для создания наших данных и выполняем печать с помощью `PrinterJob` API.

```
//файл: UnbelievablySimplePrint.java
import java.awt.*;
import java.awt.print.*;
public class UnbelievablySimplePrint implements Printable
{
private static Font sFont = new Font("Serif", Font.PLAIN , 64);
public int print(Graphics g, PageFormat Pf, int pageIndex)
throws PrinterException
{
if (pageIndex > 0) return NO_SUCH_PAGE;
Graphics2D g2 = (Graphics2D)g;
g2.setFont(sFont);
g2.setPaint(Color.black);
g2.drawString("Save a tree!", 96, 144);
return PAGE_EXISTS;
}
public static void main(String[] args) {
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(new UnbelievablySimplePrint());
if (job.printDialog()) {
```

```
try {
job.print();
}
catch (PrinterException e) {}
}
System.exit(0);
}
}
```

Это простой пример. Мы создали реализацию интерфейса `Printable`, названную `UnbelievablySimplePrint`. Это очень простой метод `print()`, который рисует некий текст.

Остальная часть работы заключена в методе `main()` и связана с настройками задания печати. Во-первых, мы создаем новый объект `PrinterJob` и сообщаем ему, что мы хотим напечатать:

```
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(new UnbelievablySimplePrint());
```

Далее мы используем метод `printDialog()`, чтобы показать стандартное диалоговое окно печати. Если пользователь нажимает кнопку `OK`, метод `printDialog()` возвращает значение true и метод `main()` переходит к печати.

Заметьте, что метод `print()` мы выполняем в знакомом составе от класса Graphics до класса `Graphics2D`. Для заданий печати доступны все возможности 2D API. В настоящих приложениях у вас, возможно, есть некоторый подкласс класса `Component`, также являющийся `Printable`. Для компонента, выполняющего схожую визуализацию на экране и на странице печати, метод `print()` может упростить вызов метода компонента `paint()`.

Глава 21

РАБОТА С ИЗОБРАЖЕНИЯМИ И ДРУГИМИ МЕДИАДАННЫМИ

До этого момента мы ограничивались работой с высокоуровневыми командами класса Graphics2D, выводя на экран статичное изображение. В этой части мы проясним все неочевидные моменты, связанные с растровой графикой, и увидим, как создаются и трансформируются изображения. Обработка изображений и их свойств производится классами пакета `java.awt.image`; на рис. 21.1 изображены важные классы из этого пакета.

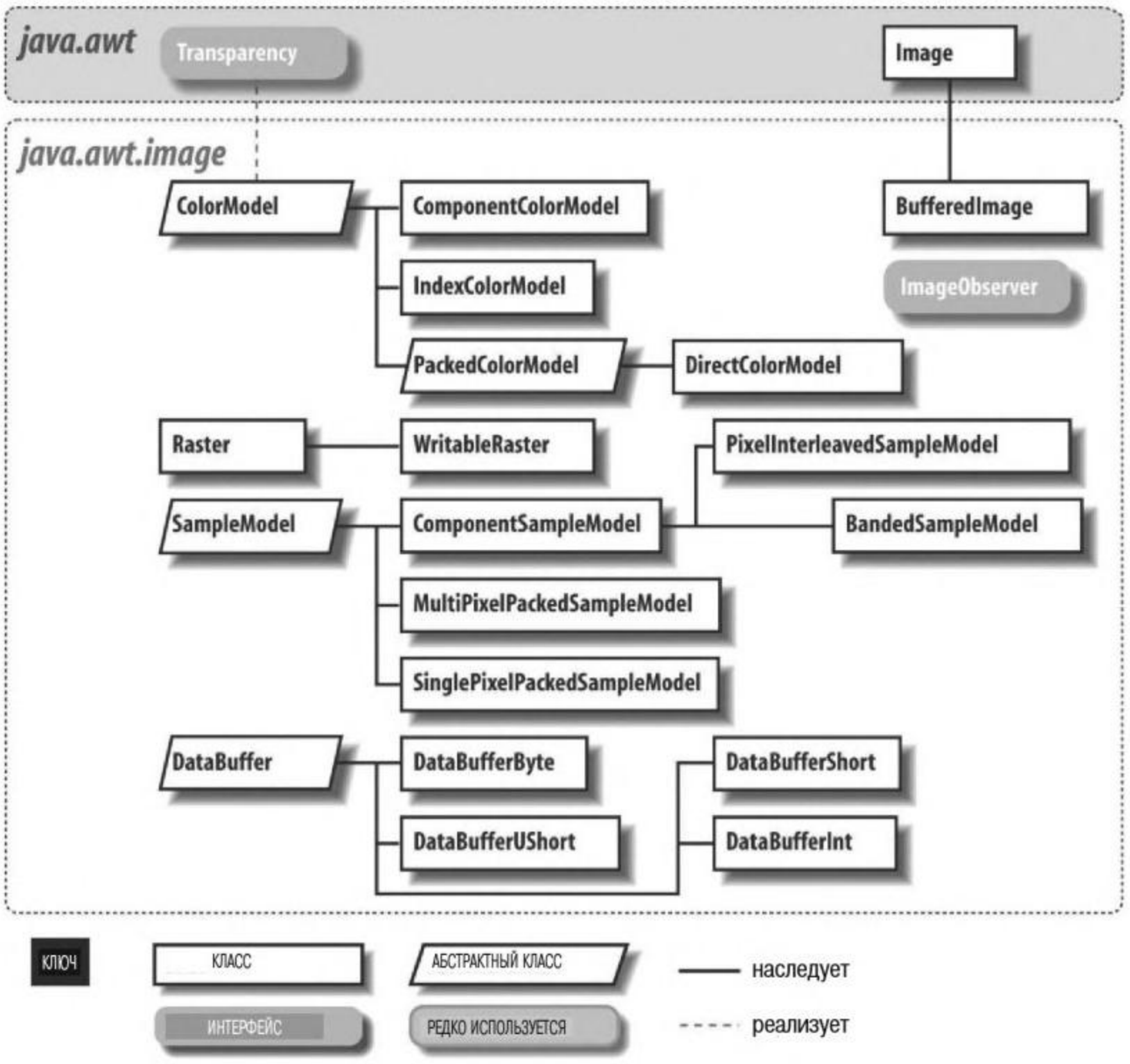

Рис. 21.1. Пакет `java.awt.image`

Во-первых, мы вернемся к обсуждению загрузки изображения и увидим, каким образом можно получить больший контроль над данными изображения, используя ImageObserver, чтобы наблюдать асинхронную обработку компонентами GUI. Также мы увидим, как использовать класс `MediaTracker` для детальной обработки. Далее мы взглянем на Buffered Image — это пиксельные данные изображения, которые передаются вам через буфер памяти. В этой главе вы познакомитесь с основами создания сложной графики, изображений или видео.

С другой стороны, мы иногда будем обращаться к пакету `java.imageio`, который является частью интерфейса Java Advanced Imaging API (JAI). Обратите внимание на JAI, если вам необходимо больше дополнительных возможностей, таких, как замощение, загрузка масштабированной версии изображения по сети, отложенное выполнение обработки изображения для работы с действительно большими изображениями.

Загрузка изображений

Одна из проблем при создании программного обеспечения для сетевых приложений заключается в том, что данные не всегда мгновенно доступны. В интернет-приложения, такие как веб-браузеры, включены части ядра Java, поэтому в обработчики изображений API была специально заложена возможность предоставления информации о ходе загрузки изображения, так как требуется некоторое время на загрузку изображения по медленной сети. Хотя большинство клиентских приложений не требуют подобной обработки изображений, все же полезно понимать этот механизм, хотя бы по той причине, что так или иначе он появляется в большинстве основных API, обрабатывающих изображения. Инструментарий библиотеки Swing добавляет собственный слой обработки изображения поверх существующего, используя такие компоненты, как `ImageIcon`, которые инкапсулируют источник изображения. После прочтения этой главы, вы будете понимать, как эти слои сочетаются друг с другом.

Интерфейс `ImageObserver`

В предыдущей главе мы упоминали, что все операции с данными изображения (к примеру, загрузка, рисование, масштабирование) позволяют указывать контролер изображения в качестве участника про-

цесса. Контролер изображения реализуется посредством интерфейса `ImageObserver`, позволяя получать уведомления при поступлении информации о загружаемом изображении. По сути, контролер изображения осуществляет обратный вызов для постепенного получения уведомлений в процессе загрузки изображения. Для статических изображений в GIF- или JPEG-форматах, контролер получает уведомления в моменты поступления фрагментов изображения и завершения загрузки изображения. Для анимации или видео (к примеру, GIF89 формат), контролер получает уведомления в конце каждого кадра, из которых создается непрерывный поток обрабатываемых пикселов.

Используя контролер изображения, можно различными способами обрабатывать полученное изображение. К примеру, в последней главе используется контролер класса `Component`. Возможно, вы не замечали работы контролера в наших примерах, однако он запускается при вызове метода `repaint()` каждый раз, когда новый фрагмент изображения становится доступным, поэтому изображения, которым необходимо продолжительное время для загрузки, могут отображаться постепенно. Существуют различные типы контролеров изображения. Одни из них дожидаются полной загрузки изображения, перед тем как передать его на отображение. Другие могут сообщать, как далеко прошел процесс загрузки.

```
Контролер изображения должен реализовать метод imageUpdate(),
определенный в интерфейсе java.awt.image.ImageObserver:
public boolean imageUpdate(Image image, int flags, int x, int y,
int width, int height)
```

Метод `imageUpdate()` вызывается графической системой, по необходимости, чтобы передать контролеру полученную информацию об изображении, в соответствующем представлении. Параметр `image` — это объект класса `Image` (загружаемое изображение). `flags` — целое число, биты которого определяют, какая информация об изображении доступна в данный момент. Значения `flags` определены, как статические (`static`) переменные интерфейса `ImageObserver`, как показано ниже:

```
//файл: ObserImageLoad.java
import java.awt.*;
import java.awt.image.*;
public class ObserveImageLoad {
   public static void main( String [] args)
   {
```

```
ImageObserver myObserver = new ImageObserver() {
   public boolean imageUpdate(
      Image image, int flags, int x, int y, int width,
         int height)
   {
      if ( (flags & HEIGHT) !=0 )
         System.out.println("Высота изображения = " + height );
      if ( (flags & WIDTH ) !=0 )
         System.out.println("Ширина изображения = " + width );
      if ( (flags & FRAMEBITS) != 0 )
         System.out.println("Получен еще один кадр!");
      if ( (flags & SOMEBITS) != 0 )
         System.out.println("Фрагмент изображения:"
            + new Rectangle( x, y, width, height ) );
      if ( (flags & ALLBITS) != 0 )
         System.out.println("Изображение получено!");
      if ( (flags & ABORT) != 0 )
         System.out.println("Загрузка изображения прервана...");
      return true;
   }
};
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image img = toolkit.getImage( args[0] );
toolkit.prepareImage( img, -1, -1, myObserver );
}
}
```

Запустите описанный выше пример, указывая файл изображения в качестве аргумента командной строки, и наблюдайте за результатом. Вы увидите увеличивающееся количество сообщений о загрузке изображения.

Целочисленный параметр `flags` определяет, каким образом интерпретировать остальные параметры: `x`, `y`, `width` и `height`. Для проверки, какое конкретно значение установлено в `flags`, мы вынуждены прибегать к некоторым бинарным операциям — использовать оператор `&(И)`. Параметры `width` и `height` могут обрабатываться по-разному. Если установлен флаг `SOMEBIT`, они представляют размер уже отправленного фрагмента изображения. Если установлены флаги `HEIGHT` или `WIDTH`, то они представляют размеры всего изображения. Наконец, метод `imageUpdate()` возвращает булево значение, которое показывает, есть или нет необходимость в дальнейшем обновлении.

В этом примере после получения объекта `Image` методом `getImage()` мы запускаем процесс загрузки, используя метод `prepareImage()` класса `Toolkit`, который передает наше изображение контролеру, в качестве аргумента. Для начала операции загрузки также достаточно вызвать методы класса `Image API drawImage()`, `scaleImage()` или запросить размеры изображения, используя методы `getWidth()`, `getHeight()`. Помните, что даже метод `getImage()`, создавший объект изображения, не начнет загрузку изображения, пока не потребуется выполнить одну из операций над ним.

Рассмотренный пример показывает основной механизм нижнего уровня для запуска и мониторинга процесса по загрузке обрабатываемого изображения. Теперь вам должно быть ясно, каким образом мы могли бы реализовать все виды сложных схем загрузки изображения и ее мониторинга. Мы будем использовать две наиболее важные стратегии: отрисовывать изображение постепенно в процессе его загрузки или отрисовывать его полностью после завершения загрузки. Выше мы рассмотрели первую схему, реализованную на основе класса `Component`. В следующей части мы рассмотрим другой класс `java.awt.MediaTracker` — обычную программу, отслеживающую процесс загрузки нескольких изображений или других медиаданных.

Класс `MediaTracker`

Если мы хотим дождаться завершения загрузки одного или группы изображений прежде, чем отобразить их, то служебный класс `java.awt.MediaTracker` может упростить нашу задачу. Класс `MediaTracker` отслеживает процесс загрузки одного или группы изображений, что позволяет нам периодически проверять состояние загрузки или дожидаться ее завершения. Класс `MediaTracker` реализует интерфейс `ImageObserver`, который мы уже рассматривали, позволяя ему получать обновления изображения.

Следующий фрагмент кода показывает, как использовать класс `MediaTracker`, чтобы дождаться готовности изображения:

```
//файл: StatusImage.java
import java.awt.*;
import javax.swing.*;
public class StatusImage extends JComponent
{
   boolean loaded = false;
   String message = "Загрузка...";
```

```
Image image;
public StatusImage( Image image ) { this.image = image; }
public void paint(Graphics g) {
   if (loaded)
      g.drawImage(image, 0, 0, this);
   else {
      g.drawRect(0,0,getSize().width - 1,getSize().height - 1);
   g.drawString(message, 20, 20);
   }
}
public void loaded() {
   loaded = true;
   repaint();
}
public void setMessage( String msg ) {
   message = msg;
   repaint();
}
public static void main( String [] args ) {
   JFrame frame = new JFrame("TrackImage");
   Image image = Toolkit.getDefaultToolkit().getImage(args[0]);
   StatusImage statusImage = new StatusImage( image );
   frame.add( statusImage );
   frame.setSize(300,300);
   frame.setVisible(true);
   MediaTracker tracker = new MediaTracker( statusImage );
   int MAIN_IMAGE = 0;
   tracker.addImage( image, MAIN_IMAGE );
      try {
         tracker.waitForID( MAIN_IMAGE ); }
      catch (InterruptedException e) {}
      if ( tracker.isErrorID( MAIN_IMAGE ) )
         statusImage.setMessage( "Ошибка!" );
      else
         statusImage.loaded();
   }
}
```

В этом примере мы создали тривиальный компонент с именем StatusImage, который получает изображение и выводит текстовое сообщение состояния, пока изображение загружается. После он отобра-

жает изображение. Мы использовали класс `MediaTracker` для загрузки изображения, упростив логику нашей реализации.

Для начала мы создали объект класса `MediaTracker` для управления изображением. Конструктор класса `MediaTracker` принимает объект класса `Component` в качестве аргумента; предполагается, что это будет компонент, на котором позже будет отрисовано изображение. Этот аргумент — пережиток первых версий Java, работающих с пакетом AWT. Если у вас нет компонента, на который удобно ссылаться, вы можете создать обычный компонент для ссылки, как показано ниже:

```
Component comp = new Component();
```

После создания объекта класса `MediaTracker` мы передаем ему изображения для управления. Каждое изображение связано с целочисленным идентификатором, который в последствии мы можем использовать для проверки состояния загрузки изображений либо для ожидания ее завершения. Несколько изображений могут быть связаны с одним идентификатором, что позволяет нам управлять ими как группой. Также значение идентификатора значимо при определении приоритета загрузки, в случае, если загружается несколько изображений, меньшее значение идентификатора имеет более высокий приоритет. В данном примере мы собираемся управлять только одним изображением, поэтому мы создали единственный идентификатор `MAIN_IMAGE` и передали его в качестве идентификатора для нашего изображения при вызове метода `addImage()`.

Далее, мы вызываем метод `waitforID()` класса `MediaTracker`, который устанавливает блокировку на изображении до завершения его загрузки. В случае успешного завершения загрузки мы указываем компоненту использовать изображение и перерисовать его. Другой элемент класса `MediaTracker` — метод `waitForAll()` ожидает загрузки всех изображений, а не только с одинаковым ID. В этом месте можно создать прерывание с помощью исключения `InterruptedException`. Кроме того, нам необходимо, используя метод `isErrorID()`, проверять наличие ошибок в процессе подготовке изображения. В нашем примере, если обнаружена ошибка, мы меняем сообщение состояния.

Методы `CheckId()` и `checkAll()` класса `MediaTracker` могут быть использованы для периодической проверки состояния загрузки изображений, возвращая значения `true` или `false`, чтобы сообщить, завершена загрузка или нет. Метод `checkAll()` выполняет проверку всех загружае-

мых изображений. Кроме того, методы `StatusId()` и `statusAll()` возвращают константу, текущее состояние загрузки изображения. Константа может принимать одно из значений определенных в классе `MediaTracker`: `LOADING` (загрузка), `ABORTED` (прервано), `ERROR` (ошибка), или `COMPLETE` (завершено). Возвращаемое значение метода `statusAll()`является побитовым `OR` состояний всех загружаемых изображений.

Может показаться, что требуется слишком много работы только для того, чтобы получить сообщение о состоянии загрузки единственного изображения. Класс `MediaTracker` более полезен, когда вы работаете со множеством необработанных изображений, которые должны быть доступны, перед тем как вы начнете выполнять следующую часть вашего приложения. Этот класс позволяет сэкономить время реализации типового интерфейса `ImageObserver` в каждом приложении. Кроме того, добавляя компонент `ImageIcon` к использованию класса `MediaTracker`, вы можете еще больше упростить работу приложений библиотеки Swing. Как это сделать, описано далее.

Класс `ImageIcon`

В главе 17 мы обсуждали библиотеку Swing, которая может работать с изображениями, используя интерфейс `Icon`. В частности, класс `ImageIcon` получает путь к файлу изображения или его `URL` и может отобразить полученное изображение на компоненте. По сути дела, `ImageIcon` использует класс `MediaTracker` для полной загрузки изображения, которое было указано в вызове его конструктора. Этот класс также может возвращать ссылку на объект `Image`. Таким образом, кратчайший способ реализовать то, что мы делали в нескольких предыдущих частях (дожидались завершения загрузки изображения перед его использованием), выглядит следующим образом:

```
ImageIcon icon = new ImageIcon("myimage.jpg");
Image image = icon.getImage();
```

Такой причудливый подход позволит сэкономить несколько строк кода, но использует объект `icon` странным и не вполне понятным образом. Кроме того, класс `ImageIcon` предоставляет вам прямой доступ к классу `MediaTracker`, используя метод `getMediaTracker()` или сообщает вам о состоянии загрузки методом `getImageLoadStatus()`. Указанные методы возвращают одну из констант, определенных в классе `MediaTracker`: `ABORTED`, `ERROR`, или `COMPLETE`.

Класс `ImageIO`

Как мы упоминали во вступлении к главе 1, пакет `javax.imageio` является стандартным расширением для работы с чтением и записью многих форматов изображений. Этот пакет является частью более крупной библиотеки Java Advanced Imaging (JAI) API. В ней есть дополнительные способы отображения и управления изображениями. В отличии от пакета AWT, имеющего относительно фиксированный набор функциональности, JAI является расширяемым фреймворком, который поддерживает подключаемые модули для новых форматов изображений и их свойств. Пакет `imageio` как часть библиотеки JAI входит в Java, начиная с версии 1.4, таким образом, мы можем использовать его преимущества в любой текущей версии Java. Класс `ImageIO` эффективно реализует API-интерфейсы, мы говорили об этом, рассматривая загрузку нового обрабатываемого изображения и ее мониторинг. Мы рассмотрим его лишь вкратце по нескольким причинам. Во-первых, он необычайно прост в использовании. Во-вторых, класс `ImageIO` изначально работает с объектами класса `BufferedImages`, а не напрямую с объектами класса `Images` AWT-интерфейса. В оставшейся части главы мы расскажем о том, как буферизированные изображения могут предоставлять свои пикселы для чтения и обработки. Наконец, класс `ImageIO` позволяет вам выполнять операции чтения и записи в файл объектов класса `BufferedImages`. В самом ядре пакета AWT нет инструментов для сохранения обрабатываемых изображений в нужных форматах в файл. Прежде всего, мы покажем, как просто загрузить изображение, используя статический метод `read()` класса `ImageIO`, который получает изображение в виде объектов одного из классов `FILE`, `URL` или `InputStream`:

```
File file = new File("/Users/pat/images/boojum.gif");

BufferedImage bi = ImageIO.read (file);
```

В этом примере мы явно указали, что метод `read()` возвращает объект класса `BufferedImage`, являющийся подтипом `Image`. Метод `ImageIO.read()`, как и метод `getImage()` AWT-интерфейса, автоматически определяет формат изображения и декодирует его должным образом. Так как класс `ImageIO` расширяемый, то полезно иметь полный список поддерживаемых им форматов изображений. Вы можете получить эту информацию, используя метод `ImageIO.getReaderFormatNames()`, который возвращает массив строк, грубо соответствующий расширению файлов, которые поддерживает класс. Класс `ImageIO` определяет формат файлов не по расширению файла, а по его содержимому.

Метод `ImageIO.read()` возвращает только полностью загруженное изображение, блокируя его в процессе загрузки. Если вам необходима более подробная информация о процессе загрузки изображения, вы можете использовать интерфейс `IIOReadProgressListener` пакета `javax.imageio.event`, который примерно соответствует `ImageObserver` AWT-интерфейсов. Чтобы использовать его, вы должны больше узнать об API-интерфейсе `ImageIO`, сперва взглянув на класс `ImageReader`, с помощью которого регистрируется слушатель:

```
import javax.imageio.*;
import javax.imageio.stream.*;
import javax.imageio.event.*;
File file = new File("image.jpg");
ImageInputStream iis = ImageIO.createImageInputStream( file );
Iterator readers = ImageIO.getImageReaders( iis );
ImageReader reader = (ImageReader)readers.next(); // выбираем следующее
изображение
reader.addIIOReadProgressListener( readProgressListener );
reader.setInput( iis, true );
BufferedImage bi = reader.read( 0/*index*/ );
```

Это очень простой код. Интерфейс `ReadProgressListener` используется, как и любой другой интерфейс событий библиотек AWT или Swing, которые мы рассматривали ранее. Вы можете обратиться к Javadoc для получения всех методов, которые вы должны реализовать.

В завершение, существует два других интерфейса, позволяющих отслеживать состояние выполнения (в дополнение к описанному выше): `IIOReadUpdateListener` и `IIOReadWarningListener`; они собирают информацию об изменениях пикселов (например, в процессе загрузки) и ошибках загрузки, соответственно. Конечно, у них есть версии с префиксом «Write», которые работают в обратном направлении, сохраняя изображение. Мы вернемся к этой теме позже в текущей главе.

Создание растровых данных

Существует два подхода к генерированию изображения. Первый, высокоуровневый способ заключается в том, что изображение используется в качестве холста для рисования, поверх которого размещаются различные объекты (с помощью методов класса `Graphics2D`). Другой подход: обрабатывать биты, которые представляют пикселы обрабаты-

ваемого изображения, самостоятельно. Этот подход сложнее, но дает вам полный контроль при обработке специфических форматов, математическом анализе или при создании обрабатываемого изображения.

Создание анимации

Начнем с более простого подхода: визуализируя изображение путем рисования. Чтобы сделать изучение более интересным, создадим анимацию. Каждый кадр мы будем передавать отдельно. Такой подход очень похож на метод двойной буферизации, который мы рассматривали в предыдущей главе, за исключением того, что в этот раз сигналом для создания следующего кадра будет не событие мыши, а таймер.

Библиотека Swing предоставляет автоматическую двойную буферизацию, то есть нет необходимости беспокоиться о мерцании анимации. Кроме того, это выглядит, как будто мы рисуем непосредственно на экране, хотя в действительности мы рисуем в изображении, которое Swing использует для двойной буферизации. Все что от нас требуется — рисовать нужную вещь в нужное время.

Рассмотрим пример, Hypnosis, который иллюстрирует технику. В этом примере отображается постоянно смещающаяся форма, которая скачет внутри компонента. В период популярности экранных заставок такой материал был действительно востребован. На рис. 21.2 показано приложение Hypnosis.



Рис. 21.2. Простая анимация

Исходный код:

```
//файл: Hypnosis.java
import java.awt.*;
import java.awt.event.*;
```

```
import java.awt.geom.GeneralPath;
import javax.swing.*;
public class Hypnosis extends JComponent implements Runnable {
   private int[] coordinates;
   private int[] deltas;
   private Paint paint;
   public Hypnosis(int numberOfSegments) {
      int numberOfCoordinates = numberOfSegments * 4 + 2;
      coordinates = new int[numberOfCoordinates];
      deltas = new int[numberOfCoordinates];
      for (int i = 0 ; i < numberOfCoordinates; i++) {
         coordinates[i] = (int)(Math.random() * 300);
         deltas[i] = (int)(Math.random() * 4 + 3);
         if (deltas[i] > 4) deltas[i] = -(deltas[i] - 3);
         }
      paint = new GradientPaint(0, 0, Color.BLUE,
                     20, 10, Color.RED, true);
      Thread t = new Thread(this);
      t.start();
      }
   public void run() {
      try {
         while (true) {
            timeStep();
            repaint();
            Thread.sleep(1000 / 24);
            }
         }
      catch (InterruptedException ie) {}
      }
   public void paint(Graphics g) {
      Graphics2D g2 = (Graphics2D)g;
   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);
      Shape s = createShape();
      g2.setPaint(paint);
      g2.fill(s);
      g2.setPaint(Color.WHITE);
      g2.draw(s);
      }
   private void timeStep() {
```

```
        Dimension d = getSize();
        if (d.width == 0 || d.height == 0) return;
        for (int i = 0; i < coordinates.length; i++) {
            coordinates[i] += deltas[i];
            int limit = (i % 2 == 0) ? d.width : d.height;
            if (coordinates[i] < 0) {
                coordinates[i] = 0;
                deltas[i] = -deltas[i];
                }
            else if (coordinates[i] > limit) {
                coordinates[i] = limit - 1;
                deltas[i] = -deltas[i];
                }
            }
        }
    private Shape createShape() {
        GeneralPath path = new GeneralPath();
        path.moveTo(coordinates[0], coordinates[1]);
        for (int i = 2; i < coordinates.length; i += 4)
            path.quadTo(coordinates[i], coordinates[i + 1],
                coordinates[i + 2], coordinates[i + 3]);
        path.closePath();
        return path;
        }
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hypnosis");
        frame.add( new Hypnosis(4) );
        frame.setSize(300, 300);
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setVisible(true);
        }
    }
```

Метод `main()` выполняет обычную подготовительную работу по настройке объекта `JFrame`, который хранит компонент нашей анимации.

Компонент `Hypnosis` использует очень простую стратегию анимации. В переменных класса `coordinates` хранится некоторое количество координатных пар. Соответствующий массив `deltas` хранит значения «дельта» (значение смещения координат), которые прибавляются к координатам каждый раз, когда предполагается изменить форму. Чтобы отобразить сложную форму, которую вы видите на рис. 21.2, `Hypnosis`

создает специальный объект Shape из массива координат каждый раз при рисовании компонента.

Конструктор класса Hypnosis выполняет две важные задачи. Во-первых, заполняет случайными значениями массивы coordinates и deltas. Количество элементов массивов определяется аргументом конструктора. Во-вторых, конструктор запускает новый поток, выполняющий анимацию.

Анимация выполняется в методе run(). Этот метод обращается к методу timeStep(), затем перерисовывает компонент и ждет некоторое время (об этом чуть позже). При каждом вызове метода timeStep() обновляется массив координат, затем вызывается метод repaint(). Результат этих операций используется при вызове метода paint(), который создает из массива координат форму и отрисовывает ее.

Метод paint() относительно прост. Он использует вспомогательный метод createShape() для создания формы из массива координат. Затем форма заполняется с использованием переменной класса Paint. Контур формы также рисуется белым.

Метод timeStep() обновляет все элементы массива координат, прибавляя значения дельта к соответствующим элементам. Если любая из координат выходит за пределы компонента, ей присваивается значение координат границы и соответствующему значению дельта присваивается противоположный знак. Это создает эффект отскакивания формы от границ компонента.

Метод createShape() создает форму из массива координат. Метод применяет класс GeneralPath, польза создания объекта Shape в том, что с его помощью можете создавать формы, используя сегменты прямых и криволинейных линий. В данном примере мы создаем форму из последовательности квадратичных кривых, замыкая их для создания области, и заполняем ее.

Анатомия класса BufferedImage

Мы так долго говорили об объектах класса java.awt.Images, о том, как они могут быть загружены и нарисованы. А что делать, если вы хотите изменить и обновить обрабатываемое изображение изнутри? Вам придется использовать более сложный тип изображений: java.awt.image.BufferedImage. Эти классы тесно связаны: класс BufferedImage,

фактически, является подклассом класса `Image`. Класс `BufferedImage` предоставляет вам весь спектр возможностей управления актуальными данными, которые составляют изображение, и больше возможностей, чем класс `Image`. Так как это потомок класса `Image`, вы можете применять к объектам класса `BufferedImage` любые методы класса `Graphics2D`, которые принимают объект `Image`. Почему не все изображения имеют тип `BufferedImages`? Потому что класс `BufferedImage` отличает повышенное потребление памяти.

Чтобы создать изображение из исходных данных, вам необходимо четко понимать, как класс `BufferedImage` собирает их вместе. Разобраться во всех деталях может быть непросто, ведь класс `BufferedImage` был разработан для поддержки изображений практически в любых форматах. Но его не так сложно использовать для выполнения стандартных операций. На рис. 21.3 представлены элементы класса `BufferedImage`.

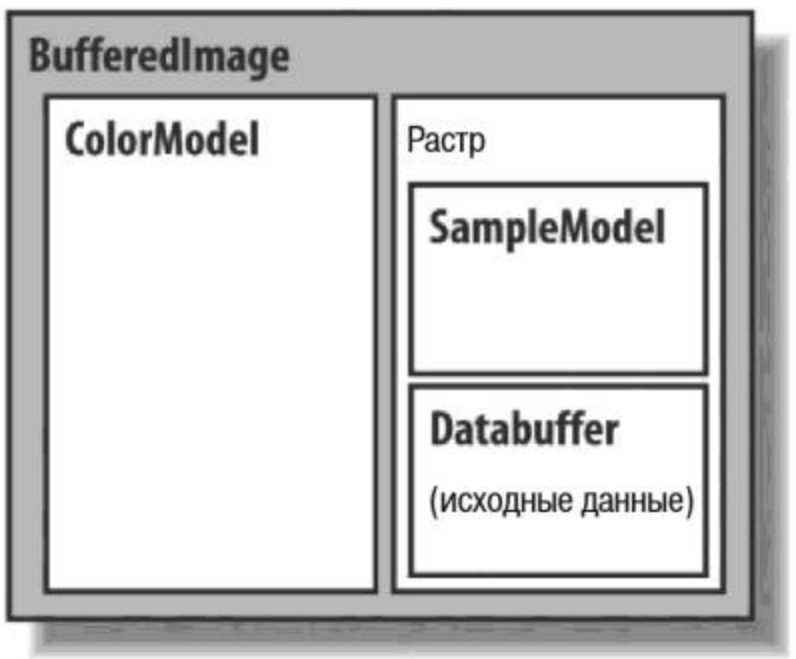

Рис. 21.3. Внутри класса `BufferedImage`

Чтобы разобраться, достаточно простого изображения: прямоугольника с цветными пикселами. С классом `BufferedImage` связано множество тонких нюансов, потому что существует целый ряд различных способов передачи цвета пикселов. Например, у вас может быть RGB-изображение, информация о цвете которого хранится в байтовом массиве, и каждый элемент этого массива содержит значение цветовых компонентов: красного, зеленого и синего. Или у вас может быть RGB-изображение, цветовые компоненты которого хранятся в виде целых чисел. Или 16-цветное изображение (оттенки серого), восемь пикселов которого хранятся в каждом элементе целочисленного массива. Существует много различных способов хранения изображений, и класс `BufferedImage` поддерживает их все.

Класс BufferedImage стоит из двух подкласов: Raster и ColorModel. Raster содержит актуальное обрабатываемое изображение. Его можно представить как массив значений пикселов изображения. Он может ответить на вопрос: «Какой цвет у пиксела 51, 17?» Для RGB-изображения класс Raster должен возвращать три значения, в то время как для изображения оттенков серого класс Raster должен возвращать единственное значение. WritableRaster подкласс класса Raster, также меняет значения обрабатываемых пикселов.

Задача подкласса ColorModel — интерпретировать обрабатываемое изображение как цвет. Класс ColorModel может преобразовывать поступающие из класса Raster значения в объекты класса Color. К примеру, цветовая модель RGB знает, как интерпретировать три обрабатываемых значения как компоненты цвета: красный, зеленый и синий. Модель оттенков серого может интерпретировать единственное значение как уровень серого. Вкратце вот как изображение показывается на экране. Графическая система получает данные о каждом пикселе из класса Raster. Далее класс ColorModel сообщает, какой цвет должен быть у каждого пиксела, и графическая система устанавливает соответствующий цвет каждому пикселу.

Класс Raster также состоит из двух подклассов: DataBuffer и SampleModel. Класс DataBuffer — это обертка для необработанных массивов данных: битовых, коротких и целочисленных. У класса DataBuffer есть подклассы DataBufferByte, DataBufferShort и DataBufferInt, которые позволяют вам создавать объекты DataBuffer из массивов необработанных данных. Вы увидите пример этой техники позже, в примере StaticGenerator.

Класс SampleModel знает, каким образом извлечь обрабатываемые значения для каждого пиксела из массива DataBuffer. Он знает структуру массивов DataBuffer и, в конечном счете, отвечает на вопрос: «Какие данные у пиксела с координатами *x,y*?» С классом Sample Models немного сложнее работать, но, к счастью, вам вряд ли придется создавать или использовать его объекты. Как мы понимаем, класс Raster имеет много статических («фабричных») методов, которые создают объекты класса Raster с заданной конфигурацией, включая их в компоненты классов Data Buffers и SampleModels.

Как показано на рисунке 21.1, 2D API поставляется с различными наборами классов ColorModels, Sample Models и DataBuffers. Они служат удобными строительными блоками, которые охватывают большинство

основных форматов хранения изображений. Вам лишь изредка потребуются подклассы каких-то из этих классов для создания объектов `BufferedImage`.

Цветовые модели

Как мы уже говорили, существует множество различных способов представления информации о цвете: красный, зеленый, синий компоненты (RGB); оттенок, насыщенность, значение (HSV); оттенок, яркость, насыщенность (HLS) и другие. Дополнительно вы можете предоставлять полную информацию о цвете каждого пиксела либо указывать только индекс цветовой таблицы. Способ, которым вы представляете цвет, называется *цветовой моделью*. 2D API предоставляет инструменты для поддержки любой из существующих цветовых моделей. Здесь мы должны разделить цветовые модели на две группы: *прямые* и *индексные*.

Как вы, наверное, догадывались, вы должны определить цветовую модель до создания обрабатываемых пикселов; абстрактный класс `java.awt.image.ColorModel` представляет цветовую модель. По умолчанию, Java 2D использует прямую цветовую модель, называемую ARGB. A означает alpha, историческое название прозрачности. RGB обращается к красному, зеленому и синему компонентам цвета и собирает их вместе, чтобы составить единый комбинированный цвет. По умолчанию в ARGB-модели каждый пиксел представлен 32-битным целым числом, которое интерпретируется как четыре 8-битных поля; поля представлены в следующем порядке: альфа (прозрачность), красный, зеленый и синий компоненты цвета, как показано на рис. 21.4.

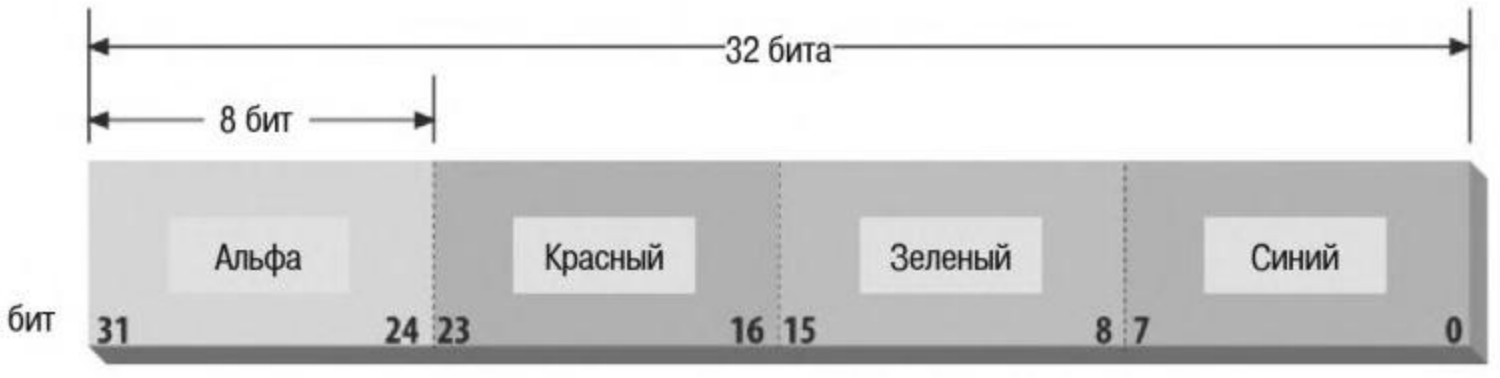

Рис. 21.4. ARGB-представление цвета

Для создания экземпляра обычной ARGB-модели вызывается статический метод `getRGBdefault()` класса `ColorModel`. Этот метод возвращает объект `DirectColorModel`; класс `DirectColorModel` — это подкласс

класса ColorModel. Вы также можете создавать другие прямые цветовые модели, используя вызов конструктора DirectColorModel, но это может пригодиться только при создании крайне необычного приложения.

В индексной цветовой модели каждый пиксел представлен меньшим количество информации: индексом в таблице вещественных цветовых значений. Несколько основных форматов изображений, включая GIF, используют индексную цветовую модель. Для некоторых приложений создание данных с использованием индексной модели может быть более удобными. Если вы пишете приложение для 8-битных или менее дисплеев, использование индексной модели может быть более эффективным, поскольку ваше аппаратное обеспечение использует индексную цветовую модель в какой-то форме.

Создание изображения

Рассмотрим создание некоего изображения. Картинка стоит тысячи слов, и, к счастью, нам понадобится куда меньше тысячи слов языка Java, чтобы создать что-то красивое. Если мы хотим лишь отобразить изображение в кадре бит за битом, мы можем сделать это очень просто, используя класс BufferedImage. Следующее приложение, ColorPan, создает изображение из целочисленного массива, хранящего пикселы в виде RGB-значений:

```
//файл: ColorPan.java
import java.awt.*;
import java.awt.image.*;
import javax.swing.*;
public class ColorPan extends JComponent {
   BufferedImage image;
   public void initialize() {
      int width = getSize().width;
      int height = getSize().height;
      int[] data = new int [width * height];
      int i = 0;
      for (int y = 0; y < height; y++) {
         int red = (y * 255) / (height - 1);
      for (int x = 0; x < width; x++) {
         int green = (x * 255) / (width - 1);
         int blue = 128;
         data[i++] = (red << 16) | (green << 8 ) | blue;
```

```
        }
    }
image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
image.setRGB(0, 0, width, height, data, 0, width);
}
public void paint(Graphics g) {
    if (image == null)
        initialize();
    g.drawImage(image, 0, 0, this);
    }
public void setBounds(int x, int y, int width, int height) {
    super.setBounds(x,y,width,height);
    initialize();
    }
public static void main(String[] args) {
    JFrame frame = new JFrame("ColorPan");
    frame.add(new ColorPan());
    frame.setSize(300, 300);
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setVisible(true);
    }
}
```

Попробуйте выполнить этот пример. Размер изображения определяется размерами окна приложения. У вас должен получиться очень красочный квадрат, который раскрашен глубоким синим в верхнем левом углу, ярко-желтым в нижнем правом углу, зеленым и красным в противоположных пределах. Мы создали объект класса `BufferedImage`, используя метод `initialize()` и затем отобразили изображение, используя метод `paint()`. Переменные значения — это одномерный массив целых чисел, которые хранят 32-битные RGB-значения для пикселов.

В методе `initialize()` мы прошли в цикле каждый пиксел изображения и назначили ему RGB-значение. Синий компонент цвета с фиксированным значением 128, половина от максимальной интенсивности. Красный компонент изменяется от 0 до 255 при изменении координаты `y` пиксела; зеленый компонент изменяется от 0 до 255 при изменении координаты `x` пиксела. Следующее выражение объединяет эти компоненты в единое значение RGB:

```
data[i++] = (red << 16) | (green << 8 ) | blue;
```

Побитовый оператор сдвига влево (<<) должен быть знаком каждому, кто программировал на C. Он просто смещает биты на некоторое указанное количество бит в нашем 32-битном значении. Когда мы создаем объект класса `BufferedImage`, все значения обнуляются. Все что мы указываем в конструкторе — это ширина, высота и формат изображения. В классе `BufferedImage` довольно много констант, представляющих форматы хранения. В данном случае мы выбрали формат `TYPE_INT_RGB`, который предполагает, что мы хотим хранить изображение как RGB-значения, представленные в виде целых. Конструктор создает соответствующие объекты классов `ColorModel`, `Raster`, `SampleModel`, и `DataBuffer`. Затем мы просто используем метод `setRGB()` для соединения наших данных с изображением. Таким образом, мы не касаемся сложных внутренностей класса `BufferedImage`. В следующий раз мы более подробно рассмотрим детали.

Теперь у нас есть изображение, и мы можем отобразить его на дисплее стандартным методом `Image()`. Мы также переопределили метод класса `Component setBounds()`, чтобы определить, когда кадр изменяет размер и инициализировать изображение с его учетом.

Обновление `BufferedImage`

Класс `BufferedImage` может быть использован для динамического обновления изображения. Имея прямой доступ к массивам обрабатываемого изображения, мы можем просто изменить значения в них и перерисовать изображение, как нам вздумается. Возможно, это простейший способ создания вашего собственного низкоуровневого программного обеспечения для анимации. Следующий пример имитирует помехи на старом черно-белом телевизионном экране. Пример создает последовательность кадров из случайных черных и белых пикселов и отображает каждый кадр, когда он готов. На рис. 21.5 показан один из кадров имитации статики.

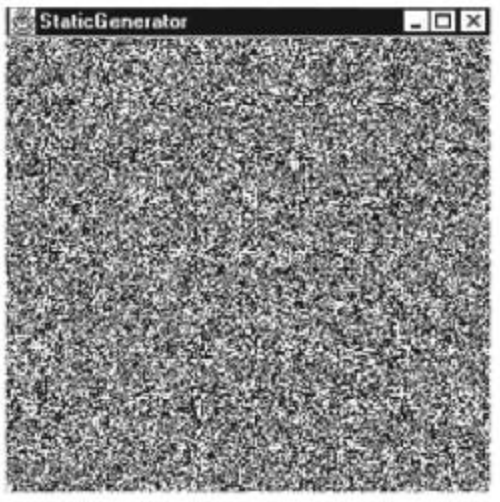

Рис. 21.5. Кадр имитации статики.

Исходный код:

```
//файл: StaticGenerator.java
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.util.Random;
import javax.swing.*;
public class StaticGenerator extends JComponent implements Runnable {
   byte[] data;
   BufferedImage image;
   Random random;
   public void initialize() {
      int w = getSize().width, h = getSize().height;
      int length = ((w + 7) * h) / 8;
      data = new byte[length];
      DataBuffer db = new DataBufferByte(data, length);
      WritableRaster wr = Raster.createPackedRaster(db, w, h, 1, null);
      ColorModel cm = new IndexColorModel(1, 2,
         new byte[] { (byte)0, (byte)255 },
         new byte[] { (byte)0, (byte)255 },
         new byte[] { (byte)0, (byte)255 });
      image = new BufferedImage(cm, wr, false, null);
      random = new Random();
      }
public void run() {
   if ( random == null )
   initialize();
   while (true) {
   random.nextBytes(data);
   repaint();
   try { Thread.sleep(1000 / 24); }
      catch( InterruptedException e ) { /* die */ }
      }
   }
public void paint(Graphics g) {
   if (image == null) initialize();
   g.drawImage(image, 0, 0, this);
   }
public void setBounds(int x, int y, int width, int height) {
   super.setBounds(x,y,width,height);
   initialize();
```

```
    }
  public static void main(String[] args) {
    //RepaintManager.currentManager(null).setDoubleBufferingEnabled(false);
    JFrame frame = new JFrame("StaticGenerator");
    StaticGenerator staticGen = new StaticGenerator();
    frame.add( staticGen );
    frame.setSize(300, 300);
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setVisible(true);
    new Thread( staticGen ).start();
    }
}
```

Метод `initialize()` настраивает объект класса `BufferedImage` так, чтобы он создавал последовательность изображений. Мы создаем это изображение последовательно, начиная с массива исходных данных. Так как мы будем отображать только два цвета, белый и черный, то нам нужен только один бит для хранения информации об одном пикселе. Пусть бит 0 означает черный цвет, и бит 1 — белый. Позже мы создадим цветовую модель, к которой будет обращаться указанный метод.

Мы храним обрабатываемое изображение в байтовом массиве, каждый элемент которого содержит 8 пикселов нашего черно-белого изображения. Длина массива вычисляется как произведение высоты и ширины изображения, разделенное на восемь. Чтобы упростить вычисление, мы условимся, что каждая строка изображения начинается с байтовой границы. Например, изображение 13 пикселов шириной использует 2 байта (16 бит) на каждую строку:

```
int length = (w + 7)/8 * h;
```

Полученное значение округляется в большую сторону, чтобы получить число байт, необходимых для хранения данных строки. Затем, умножая его на высоту изображения, мы получим количество байт, необходимое для хранения информации о всех пикселах. Далее создается необходимый байтовый массив. Переменная класса `data` хранит ссылки на этот массив. Позже мы будем использовать переменную `data` для динамического изменения обрабатываемого изображения. У нас есть массив, хранящий обрабатываемое изображение, теперь очень просто создать на его основе объект класса `DataBuffer`:

```
data = new byte[length];
DataBuffer db = new DataBufferByte(data, length);
```

Класс `DataBuffer` имеет несколько подклассов, таких как DataBuffer Byte, который упрощает создание буфера данных из нашего исходного массива.

Логически следующий шаг — создание объекта класса `SampleModel`. Затем мы могли бы создать объект класса `Raster` наследованием от классов `SampleModel` и `DataBuffer`. К счастью, класс `Raster` содержит набор полезных статических методов, которые создают стандартные типы объектов класса `Rasters`. Один из таких методов создает объект класса `Raster` из данных, элементы массива которых содержат в себе информацию о нескольких пикселах. Мы просто используем этот метод, сообщая ему данные буфера, ширину и высоту и указывая, что каждый пиксел использует один бит:

```
WritableRaster wr = Raster.createPackedRaster(db, w, h, 1, null/*ul
corner*/);
```

Последний аргумент этого метода — объект класса `java.awt.Point`, который указывает, где должен находится верхний левый угол объекта класса `Raster`. Передавая значение `NULL`, мы применяем значения по умолчанию 0, 0.

Последний кусочек пазла — класс `ColorModel`. Каждый пиксел представлен значением 0 либо 1, но как это значение должно соответствовать цвету? В этом случае мы используем объект класса `IndexColorModel` с очень маленькой палитрой. Палитра содержит только две записи, по одной для черного и белого цвета:

```
ColorModel cm = new IndexColorModel(1, 2,
new byte[] { (byte)0, (byte)255 },
new byte[] { (byte)0, (byte)255 });
```

Конструктор класса `IndexColorModel`, который мы используем, устанавливает, что одному пикселу соответствует один бит (1), количество записей в палитре — две (2), и три битовых массива, которые являются компонентами красного, зеленого и синего цветов (RGB) цветовой палитры. Наша палитра содержит два цвета: черный (0, 0, 0) и белый (255, 255, 255).

Теперь, когда мы собрали все кусочки, мы должны создать объект класса BufferedImage. Это изображение также хранится как переменная класса, так что мы сможем отобразить ее позже. Чтобы создать объект

BufferedImage, мы передаем цветовую модель и растровые записи, которые мы создали:

```
image = new BufferedImage(cm, wr, false, null);
```

Мы завершили всю сложную работу. Наш метод paint() отобразит изображение, используя метод drawImage().

Метод init() запускает поток, который создает пиксельные данные. Метод run() заботится о создании пиксельных данных. Он использует объект java.util.Random, чтобы заполнить байтовый массив обрабатываемого изображения случайными значениями. Так как массив данных — это актуальные данные для нашего изображения, то изменение значений этих данных влечет за собой изменение представления изображения. После того как мы заполнили массив случайными данными, вызываем метод repaint(), отражая новое изображение на экране.

Вы можете отключить двойную буферизацию, раскоментировав строку содержащую RepaintManager. Теперь изображение будет постоянно мерцать, сделавшись еще больше похожим на экран старого телевизора!

Вот и все. Стоит отметить, насколько просто создать анимацию. Когда у нас есть объект BufferedImage, мы обращаемся к нему как к любому другому изображению. Код, который создает последовательность изображений, может быть сколь угодно сложным. Но его сложность никогда не касается размещения изображения на экране и его обновления.

Настройка обрабатываемых изображений

Фильтр изображений — это объект, который позволяет трансформировать изображение. Интерфейс Java 2D API поддерживает настройку изображений, используя интерфейс BufferedImageOp. Фильтр изображения использует в качестве входных данных объект BufferedImage (изображение-источник) и выполняет обработку изображения, создавая другой объект BufferedImage (конечное изображение). 2D API поставляется с удобным набором инструментов реализации интерфейса BufferedImageOp, как показано в табл. 21.1.

Рассмотрим две простейших операции над изображением. Во-первых, попробуйте следующее приложение. Оно загружает изображение (первый аргумент в командной строке — имя файла) и изменяет его

различными способами, которые вы выбираете в выпадающем списке. Приложение показано на рис. 21.6.

Табл. 21.1. Операторы, работы с изображениями в 2D API

Имя	Описание
`AffineTransformOp`	Геометрическая трансформация изображения
`ColorConvertOp`	Изменение цвета на противоположный
`ConvolveOp`	Выполнение свертки, математической операции, которую можно использовать для размытия, резкости, или других подобных операций над изображениями.
`LookupOp`	Использует одну или более справочных таблиц для обработки значений изображения
`RescaleOp`	Использует умножение для обработки значений изображения

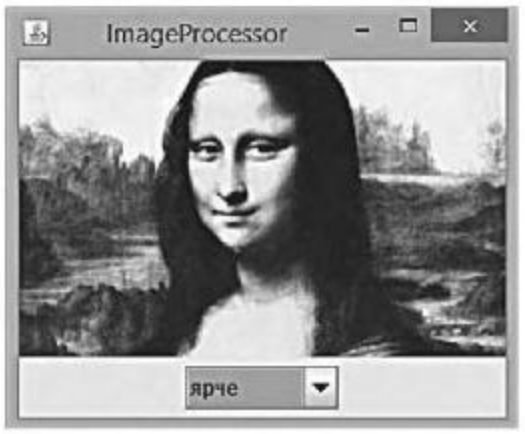

Рис. 21.6. Приложение `ImageProcessor`

Исходный код:

```
//файл: ImageProcessor.java
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;
import javax.swing.*;
public class ImageProcessor extends JComponent {
   private BufferedImage source, destination;
   private JComboBox options;
   public ImageProcessor( BufferedImage image ) {
      source = destination = image;
      setBackground(Color.white);
      setLayout(new BorderLayout());
      // создаем панель для элемента с выпадающим списком
      JPanel controls = new JPanel();
```

```
      // создаем элемент с выпадающим списком,
      // содержащий названия операций над областью
      options = new JComboBox(
   new String[] { "[источник]", "ярче", "темнее", "повернуть", "масштаб" }
   );
      // выполнение обработки при изменении выбранного значения
options.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent ie) {
      // получаем выбранную в выпадающем списке опцию
String option = (String)options.getSelectedItem();
      // обрабатываем изображение в соответствии с выбранной опцией
BufferedImageOp op = null;
      if (option.equals("[исходный]"))
         destination = source;
      else if (option.equals("ярче"))
         op = new RescaleOp(1.5f, 0, null);
      else if (option.equals("темнее"))
         op = new RescaleOp(.5f, 0, null);
      else if (option.equals("вращение"))
         op = new AffineTransformOp(
            AffineTransform.getRotateInstance(Math.PI / 6), null);
      else if (option.equals("масштаб"))
         op = new AffineTransformOp(
            AffineTransform.getScaleInstance(.5, .5), null);
      if (op != null) destination = op.filter(source, null);
         repaint();
   }
});
controls.add(options);
add(controls, BorderLayout.SOUTH);
}
public void paintComponent(Graphics g) {
int imageWidth = destination.getWidth();
int imageHeight = destination.getHeight();
int width = getSize().width;
int height = getSize().height;
g.drawImage(destination,
(width - imageWidth) / 2, (height - imageHeight) / 2, null);
}
public static void main(String[] args) {
String filename = args[0];
```

```
ImageIcon icon = new ImageIcon(filename);
Image i = icon.getImage();
// отображаем Image в BufferedImage
int w = i.getWidth(null), h = i.getHeight(null);
BufferedImage buffImage = new BufferedImage(w, h,
BufferedImage.TYPE_INT_RGB);
Graphics2D imageGraphics = buffImage.createGraphics();
imageGraphics.drawImage(i, 0, 0, null);
JFrame frame = new JFrame("ImageProcessor");
frame.add(new ImageProcessor(buffImage));
frame.setSize(buffImage.getWidth(), buffImage.getHeight());
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setVisible(true);
}
}
```

В приложении `ImageProcessor` доступно совсем немного функций. После того как вы поэкспериментируете с ним, прочтите о подробностях в следующем разделе.

Как работает приложение `ImageProcessor`

Основная операция в `ImageProcessor` очень прямая. Она загружает исходное изображение, переданное как аргумент командной строки в метод `main()`. Это изображение отображается вместе с выпадающим списком. Когда вы выбираете различные опции из выпадающего списка, `ImageProcessor` осуществляет соответствующие операции над изображением и показывает результат (конечное изображение).

Большинство из этих операций производится в обработчике события `ItemListener`, который создан конструктором `ImageProcessor` (мы используем такой сомнительный вариант, поскольку не хотим останавливаться на обработчиках событий). В зависимости от того, какая опция выбрана, объект `op` класса `BufferedImageOp` создается и используется для обработки изображения, как здесь:

```
destination = op.filter(source, null);
```

Конечное изображение возвращается методом `filter()`. Если у вас уже есть обработанное изображение верного размера для вывода, мы можем передать его вторым аргументом в метод `filter()`, который бы немного улучшил представление изображения.

Если вы передаете значение **null**, как в нашем примере, создается и возвращается соответствующее конечное изображение. Как только конечное изображение создано, у метода `paint()` остается простая задача: отрисовать конечное изображение и разместить его в центре компонента.

Преобразование объекта `Image` в объект `BufferedImage`

Обработка изображения может быть выполнена только над объектами класса `BufferedImages`, а не `Images`. Помните, что все инструменты ядра AWT работают с объектами класса `Image`, и только если вы загружаете изображение, используя пакет `ImageIO`, вы получите объекты класса `BufferedImages`. Наш пример `ImageProcessor` демонстрирует важную технику: как преобразовать простой объект `AWT Image` в объект `BufferedImage`. Вы делаете, когда рисуете в буфер, фактически копируя данные. Метод `main()` загружает объект `Image` из файла используя `Toolkit` метод `getImage()`:

```
Image i = Toolkit.getDefaultToolkit().getImage(filename);
```

Далее метод `main()` использует `MediaTracker`, чтобы убедиться, что изображение полностью загружено.

Финальный трюк преобразования объекта `Image` в объект `BufferedImage`: отрисовать объект `Image` на поверхности рисования объекта `BufferedImage`. Так как мы знаем, что объект `Image` полностью загружен, нам необходимо лишь создать объект класса `BufferedImage`, получить его графический контекст и отрисовать в нем объект `Image`:

```
BufferedImage bi = new BufferedImage(w, h,
BufferedImage.TYPE_INT_RGB);
Graphics2D imageGraphics = bi.createGraphics();
imageGraphics.drawImage(i, 0, 0, null);
```

Использование класса `RescaleOp`

Изменение насыщенности — операция над изображением, при которой каждое значение, представляющее пиксел изображения, умножается на некую константу. Эта операция ни в коем случае не влияет на размер изображения (если вы вдруг подумали, что речь идет о масштабировании), но эта операция влияет на яркость цвета пикселов.

В RGB-изображении, к примеру, каждый компонент цвета (красный, зеленый, синий) будет умножен на какой-то коэффициент. Если хотите, можете проверить результат добавляя смещение. В интерфейсе 2D API, операция `rescaling` представлена классом `java.awt.image.RescaleOp`.

Чтобы создать такой оператор, необходимо указать множитель, смещение и набор оттенков, для контроля качества изображения. В данном случае мы будем использовать нулевое смещение и не будем беспокоиться об оттенках (передавая значение `null`):

```
op = new RescaleOp(1.5f, 0, null);
```

Здесь мы установили множитель равный 1,5 и значение смещения равное 0. Все значения в конечном изображении будут в 1,5 раза превышать значения изображения источника, в результате изображение станет светлее. Для выполнения операции, мы вызываем метод `filter()` интерфейса `BufferedImageOp`.

Использование класса `AffineTransformOp`

Аффинная трансформация — это вид двумерной трансформации, при которой сохраняется параллельность линий. Это операции масштабирования, вращения и смещения. Оператор класса `java.awt.image.AffineTransformOp` геометрически изменяет изображение-источник для создания конечного изображения. Чтобы создать объект `AffineTransformOp`, определяющий нужную вам трансформацию, вам понадобится `java.awt.geom.AffineTransform`. Приложение `ImageProcessor` включает два примера этого оператора, один для вращения и один для масштабирования. Как и ранее, конструктор `AffineTransformOp` поддерживает набор дополнительных настроек, мы опять же будем передавать `null`, для упрощения:

```
else if (option.equals("rotate"))
op = new AffineTransformOp(
AffineTransform.getRotateInstance(Math.PI / 6), null);
else if (option.equals("scale"))
op = new AffineTransformOp(
AffineTransform.getScaleInstance(.5, .5), null);
```

В обоих случаях, мы получаем объект `AffineTransform`, вызывая один из его статических методов. В первом случае мы получаем трансформацию вращения на указанный угол. Эта трансформация обернута

в AffineTransformOp. В реультате этой операции мы получаем эффект вращения изображения-источника вокруг своей оси для получения конечного изображения. Во втором случае трансформация масштабирования завернута в AffineTransformOp. Два масштабирующих значения: 0,5 и 0,5, определяют что изображение должно уменьшиться вполовину по обеим осям: x и y.

При использовании AffineTransformOp для масштабирования изображения важно отметить две вещи. Быстрое масштабирование всегда приводит к ухудшению качества. При уменьшении размера (как и при любом аффинном преобразовании) вы можете выбирать между скоростью и качеством. Используя AffineTransformOp.TYPE_NEAREST_NEIGHBOR, как второй аргумент в конструкторе AffineTransformOp вы получите больше скорости. Для наилучшего качества используйте аргументы AffineTransformOp.TYPE_BICUBIC. Используя AffineTransformOp.TYPE_BILINEAR вы получите компромисс между скоростью и качеством.

Интересно заметить, что используя AffineTransformOp, вы может «потерять» часть вашего изображения при трансформации. Например, когда вы используете оператор вращения в приложении ImageProcessor, конечное изображение будет частично обрезано по сравнению с начальным. Изображение-источник и конечное изображение имеют одно происхождение, поэтому если любая часть изображения при трансформации попадает в отрицательную область координат x и y, то она теряется. Для решения проблемы, вы можете структурировать вашу трансформацию таким образом, чтобы окончательное изображение полностью находилось в положительной области координат.

Сохранение обрабатываемого изображения

Мы уделили много времени загрузке изображений из файлов, созданию и изменению обрабатываемых изображений, но ничего не сказали о их сохранении. Во-первых, о том, что сохранение изображения в файл формата JPG или GIF в действительности включает два этапа: перекодирование данных (высокий уровень сжатия данных — это способ оптимизации форматов изображений) и затем запись в файл, возможно с различными метаданными. Как мы упоминали ранее, ядро AWT не содержит инструментов для кодирования обрабатываемых изображений, только для раскодирования. В отличие от AWT, у фреймворка ImageIO

есть возможность записывать файлы изображений в любых форматах, которые он может прочитать.

Записать объект `BufferedImage` означает вызвать статический метод `write()` Фреймворка `ImageIO`:

```
File outFile = new File("/tmp/myImage.png");
ImageIO.write( bufferedImage , "png", outFile );
```

Второй аргумент — строковый идентификатор, который именует тип изображения. Вы можете получить список поддерживаемых форматов, вызвав метод `ImageIO.getWriterFormatNames()`. Мы должны заметить, что фактически тип аргумента изображения — это интерфейс, называемый `RenderedImage`, но реализует его объект `BufferedImageimplements`.

Вы можете получить больший контроль над преобразованием (например, установки качества `JPG`), передавая `ImageWriter` в выходном формате и используя `ImageWriteParams`. Этот процесс похож на отслеживание состояния выполнения, пример которого приводился в разделе «Класс `ImageIO`» ранее в этой главе.

Работа со звуком

Сейчас мы отвлечемся от изображений и переключимся на аудио. Пакет Java Sound API предоставляет широкую поддержку для создания аудиоданных и MIDI-музыки, манипуляций над ними, а также контроль над MIDI устройствами. У нас осталось место лишь для краткого обзора возможности проигрывать простые сэмплы и музыкальные файлы MIDI. Со стандартной поддержкой пакета Java Sound поставляемой с Java, вы можете проигрывать широкий спектр форматов файлов, включая: AIFF, AU, Windows WAV, стандартные MIDI-файлы, и Rich Music Format (RMF) файлы. Мы обсудим и другие форматы (такие, как MP3) наряду с видеосредствами в следующей части.

`java.applet.AudioClip` определяет простейший интерфейс для объектов, которые могут воспроизводить звук. Объект, который реализован как `AudioClip`, управляется методами: `play()` – воспроизвести звук, `stop()` – остановить воспроизведение звука или `loop()` – воспроизводить непрерывно.

Класс `Applet` предоставляет метод статической обработки `newAudio Clip()`, который извлекает звук из файла или по сети. И нет причин, по

которым мы не могли бы использовать его в других приложениях, помимо апплетов. Метод получает абсолютный или относительный URL для указания местоположения медиа-файла и возвращает его как объект `AudioClip`. Следующее приложение, `NoisyButton`, демонстрирует простой пример:

```
//файл: NoisyButton.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class NoisyButton {
public static void main(String[] args) throws Exception {
JFrame frame = new JFrame("Шумная кнопка");
java.io.File file = new java.io.File( args[0] );
final AudioClip sound = Applet.newAudioClip(file.toURL());
JButton button = new JButton("Гав!");
button.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) { sound.play(); }
});
Container content = frame.getContentPane();
content.setBackground(Color.pink);
content.setLayout(new GridBagLayout());
content.add(button);
frame.setVisible(true);
frame.setSize(200, 200);
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setVisible(true);
}
}
```

Запустите приложение `NoisyButton`, передав имя аудио файла, который вы хотите воспроизвести как аргумент. (Мы предлагаем один с названием: *bark.aiff*.) `NoisyButton` извлекает объект `AudioClip`, используя класс `File` и метод `URL()` ссылаясь на него, как на `URL`. Когда кнопка нажата, мы вызываем метод `play()` класса `AudioClip` для начала воспроизведения. После чего файл воспроизводится до того момента, как мы вызовем метод `stop()`, чтобы прервать воспроизведение.

Это простой интерфейс, но за кадром остаются очень сложные механизмы. Далее мы рассмотрим Java Media Framework, который поддерживает широкий спектр медиаданных.

Java Media Framework

Java умеет воспроизводить видео! Но для этого нам необходимо одно из стандартных Java API расширений: Java Media Framework (JMF). JMF определяет набор интерфейсов и классов в пакетах `javax.media` и `javax.media.protocol`. Вы можете загрузить последнюю версию JMF с сайта Oracle. Для использования JMF добавьте *jmf.jar* в директорию classpath. Или, в зависимости от того, какую версию JMF вы скачали, программа инсталляции может сделать это за вас.

Мы лишь поверхностно рассмотрим JMF, поработав с важным интерфейсом, называемым `Player`. Конкретные реализации Player связаны с различными типами медиа-контейнеров: Apple QuickTime (*.mov*) и Windows Video (*.avi*). Полный список поддерживаемых типов мультимедийных файлов и кодеков вы можете получить в документации к JMF. Существуют также проигрыватели для аудиофайлов, включая формат MP3. Проигрыватели обрабатываются высокоуровневыми классами в JMF называемыми `Manager`. Один из способов реализовать `Player` — указать URL-адрес видеофайла.

Как насчет стандартного формата, известного как MP3, который используется в проигрывателе Windows Media?

```
Player player = Manager.createPlayer(url);
```

Из-за того, что видеофайлы имеют большой размер и воспроизведение их требует значительных системных ресурсов, проигрыватели реализуют многоэтапный жизненный цикл, с того момента, как они создаются, до того момента, как они начинают что-то воспроизводить. Мы рассматриваем только один шаг — реализацию. На этом шаге проигрыватель определяет (обращаясь к медиафайлу), какие системные ресурсы необходимы, чтобы воспроизвести данный медиафайл.

```
player.realize();
```

Метод `realize()` сразу возвращает значение; он перемещает выполняющийся процесс в отдельный поток. Когда объект `Player` заканчивает свою работу, он высылает событие. Когда вы получите это событие, вы можете получить один из двух объектов `Components` из `Player`. Во-первых, визуальный компонент, для визуализации медиатипов, показывающих медиа. Во вторых, управляющий компонент, который предоставляет сборный пользовательский интерфейс для контроля презентации медиа. Обычно управление представлено кнопками воспроизведения,

остановки и паузы, а также регулятором громкости и некоторыми дополнительными возможностями.

`Player` должен быть реализован до того, как вы запросите данные компоненты, так как они содержат важную информацию, например насколько большим должен быть компонент. После этого обратиться к компоненту просто.

Пример:

```
Component c = player.getVisualComponent();
```

Сейчас нам необходимо лишь разместить компоненты где-либо на экране. Мы можем воспроизводить медиа где угодно (хотя в действительности это несколько раз изменяет внутреннее состояние объекта `Player`):

```
player.start();
```

Следующий пример, `MediaPlayer`, использующий JMF для загрузки и отображения видео или аудио файла из определенного URL:

```
//файл: MediaPlayer.java
import java.awt.*;
import java.net.URL;
import javax.swing.*;
import javax.media.*;
public class MediaPlayer
{
public static void main( String[] args ) throws Exception {
final JFrame frame = new JFrame("MediaPlayer");
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
URL url = new URL( args[0] );
final Player player = Manager.createPlayer( url );
player.addControllerListener( new ControllerListener()    {
public void controllerUpdate( ControllerEvent ce ) {
if ( ce instanceof RealizeCompleteEvent )
{
Component visual = player.getVisualComponent();
Component control = player.getControlPanelComponent();
if ( visual != null )
frame.add(visual, BorderLayout.CENTER );
frame.add( control, BorderLayout.SOUTH );
```

```
frame.pack();
frame.setVisible( true );
player.start();
}
}
});
player.realize();
}
}
```

Этот класс создает `JFrame`, который хранит медиа. Далее он создает объект `Player`, используя URL определенный в командной строке, и сообщает `Player` метод `realize()`. Мы не можем сделать больше ничего, пока `Player` выполняется, поэтому оставшийся код сработает внутри `ControllerListener`, после того как будет получен `RealizeCompleteEvent`.

В обработчике событий мы получаем визуальный проигрыватель и компоненты контролера и добавляем их в `JFrame`. Мы отображаем `JFrame` и в итоге проигрываем видео. Это очень просто!

Чтобы использовать `MediaPlayer`, укажите URL-адреса видео- или аудиофайлов в командной строке.

Ниже пара примеров:

```
% java MediaPlayer file:dancing_baby.avi
% java MediaPlayerhttp://myserver/mp3s/TheCure/KissMe/catch.mp3
```

На рис. 21.7 представлен «танцующий малыш» — AVI-файл, запущенный в `MediaPlayer`.



Рис. 21.7. Видеоролик с изображением танцующего малыша

Глава 22
JAVABEANS

JavaBeans — это и шаблон проектирования, и компонентная архитектура Java. Это набор правил для создания часто используемых элементов программного обеспечения, которые могут быть понятны как разработчикам, так и инструментам разработки. Создание компонентов в спецификации JavaBeans означает, что ваш код будет более удобочитаемым и вам не придется писать слишком много стандартного кода для объединения компонентов. Вы также можете использовать среду разработки JavaBeans, способную распознавать особенности этих компонентов автоматически и объединять их как plug-and-play для разработки частей приложений. В этой главе мы будем использовать NetBeans IDE для создания простых приложений, визуально добавляя компоненты Java bean. Конечно, мы можем создать собственные часто используемые bean-компоненты и добавить их на панель **Palette** (Палитра) IDE.

Хотя визуальная разработка приложений так и не достигла ожидаемого уровня (мы рассмотрим некоторые причины этого позже), она очень полезна в разработке GUI. Возможно, более важно понимать значение основных шаблонов JavaBeans, так как они прочно укоренились во всех аспектах ядра Java API. В данной главе мы рассмотрим все эти вопросы, включая программирование с JavaBeans вручную и некоторые связанные с ним API.

Что такое bean-компонент?

Что же такое JavaBeans? JavaBeans (архитектура) — это определенный набор правил; *компонент Java bean* — это обычный компонент Java, действующий по этим правилам. То есть, компоненты Java bean — это объекты языка Java, которые соответствуют JavaBeans API и шаблону проектирования. Таким образом, они могут распознаваться и управляться инструментами разработки приложений, в особенности визуальной, так же хорошо, как и при ручном программировании. Bean-компоненты существуют и работают в среде выполнения Java, как и все объекты Java. Они взаимодействуют со своими «соседями», используя события

и другие обычные методы обращения.Примеры компонентов Java bean мы можем найти в пакете `javax.swing`. Все известные компоненты, такие как `JButton`, `JTextArea`, `JScrollpane`, соответствуют шаблонам проектирования JavaBeans и являются bean-компонентами. То, что вы узнали в главе 16 о Swing компонентах, должно было подготовить вас к пониманию bean-компонентов. Хотя большинство Swing-компонентов не слишком полезны по отдельности, bean-компоненты также могут быть большими и сложными элементами приложений, такими как электронные таблицы и редакторы документов. Мы еще поговорим о том, что в действительности делает bean-компонент таковым. Сейчас мы хотим дать вам представление о том, как они используются.

Одна из целей JavaBeans: сделать возможной визуальную разработку приложений. Bean-компоненты могут выбираться на панели **Palette** (Палитра) и управляться графическим разработчиком приложений. Swing компоненты, упомянутые нами ранее, очевидные кандидаты на эту же роль. Но bean-компоненты могут быть не только простыми элементами пользовательского интерфейса, такими как кнопки или полосы прокрутки, но и более сложными и абстрактными составляющими. Можно подумать, что bean-компоненты — это всегда графические объекты, но они также могут реализовать любую часть приложения, включая «скрытую»: вычисления, хранение и взаимодействие. Существует три характеристики архитектуры JavaBeans, позволяющие сделать возможным такой вид работы с компонентами приложения:

Шаблоны проектирования

Наиболее важная характеристика компонентов Java bean — простой уровень стандартизации. Шаблоны (паттерны) проектирования (например, соглашения программирования) распознают инструменты и узнаваемые особенности bean-компонентов и управляют ими, не зная их реализации. Мы должны сказать, что bean-компоненты документируются самостоятельно. Исследуя bean-компонент, мы можем узнать, какие события он может создавать и получать; также мы можем изучить его свойства (эквивалентно открытым переменным) и методы. Bean-компоненты могут также предоставлять явную информацию об их функциях разработанных специально для IDE.

Рефлексия

Рефлексия — это важная функция языка Java, как уже обсуждалось в главе 7. Рефлексия создает возможность инспектировать Java код и управ-

лять новыми объектами Java в системе исполнения. В контексте JavaBeans рефлексия позволяет инструменту разработки анализировать возможности bean-компонентов, проверять значения их полей, вызывать их методы. По существу, рефлексия дает возможность объектам Java, которые выполняются при запуске, выполнять все те действия, которые могли бы быть выполнены, если если бы объекты были объединены во время компиляции. Даже если bean-компонент не поставляется вместе со «встроенной» документацией, мы все же можем собрать информацию о его возможностях и свойствах, прямо исследуя класс с помощью рефлексии.

Сериализация объектов

Наконец, Java Serialization API позволяет нам «заморозить» работающее приложение или компонент приложения и возобновить выполнение его позже. Это важная возможность, которая позволяет объединять части приложения без создания излишнего кода. Вместо настройки и компилирования большого количества Java-кода для создания нашего приложения при запуске мы можем просто настроить наши bean-компоненты, скорректировать их представления и сохранить их. Позже bean-компоненты могут быть переустановлены со всеми их состояниями и точками взаимодействия. Это делает возможным существование фундаментально разных подходов к процессу проектирования. Мы можем использовать сериализацию объектов ручного Java кода так же просто, как и смешивать сериализованные bean-компоненты с простыми старыми классами и другим Java-кодом. Также существует «долгосрочный» механизм сериализации объектов, сохраняющий компоненты Java bean в XML формате, который очень гибок в отношении изменения классов.

Состав bean-компонента

Примеры bean-компонентов, упомянутые нами ранее, варьируются от простых кнопок до электронных таблиц. Как нетрудно догадаться, bean-компонент «кнопка» куда менее сложен, чем «электронная таблица», и используется на других уровнях разработки приложений. На каком уровне bean-компоненты предназначены для использования? Архитектура JavaBeans предполагает ранжирование от малого до большого; простые bean-компоненты могут использоваться для создания сложных bean-компонентов. Небольшой bean-компонент может иметь единственный класс; а большой — много классов. Bean-компоненты также могут работать совместно, используя собственный контейнер для предоставления сервисов другим bean-компонентам.

Простые bean-компоненты нечто большее, чем обычный компонент Java. Фактически любой класс Java, который содержит конструктор по умолчанию (пустой) может считаться bean-компонентом. Bean-компонент должен быть сериализуемым, хотя описание JavaBeans не строго требует этого. Существует два критерия, гарантирующих, что мы можем создать экземпляр bean-компонента динамически и позже сохранить его как часть группы или композиции bean-компонента. Не существует других требований. Bean-компоненты не требуют наследования главного класса bean-компонента и не реализуют никаких специальных интерфейсов.

Успешный bean-компонент должен отправлять и получать события и передавать их свойства в мир. Для этого он придерживается соответствующего шаблона проектирования для именования уместных методов таким образом, чтобы эти особенности могли быть автоматически обнаружены. Большинство нетривиальных bean-компонентов предназначеных для создания в IDE визуальной разработки приложений, также предоставляют информацию о себе в форме класса `BeanInfo`. Класс `BeanInfo` реализует интерфейс `BeanInfo`, который содержит методы, описывающие функции bean-компонента более детально используя дополнения такие, как значки, показываемые пользователю. Обычно такая «информация о bean-компоненте» поставляется отдельным классом, который назван в честь этого и поставляется вместе с bean-компонентом.

NetBeans IDE

Обсуждение bean-компонентов не будет полным, если мы не уделим время теме средств разработки, в которых они используются. В этой главе мы исследуем NetBeans IDE версии 7.2, демонстрируя примеры bean-компонентов. NetBeans — популярное средство разработки. NetBeans предлагает мощные источники и возможности визуального программирования, шаблоны, которые помогают создавать различные типы Java классов, и средства компиляции, запуска и отладки приложений в одном инструменте. В частности, мы будем использовать визуальную среду разработки NetBeans, объединенную с нашими компонентами Java bean и испытывать их. Другие IDE предлагают схожие возможности. См. в приложении A краткое сравнение NetBeans и Eclipse IDE. Мы выбрали NetBeans для использования в этой главе, поскольку это развитое средство создания GUI, с ним просто представить концепцию.

Так как NetBeans — это полномасштабная продуктивная среда разработки, в ней имеется много функций, которые мы не используем в на-

ших примерах. По этой причине мы не можем поместить здесь полное ее описание. Мы ограничимся лишь краткими указаниями, чтобы продемонстрировать компоненты Java bean. Большинство техник, которые вы здесь увидите, также хорошо будут реализовываться в какой-то форме в других IDE.

Установка и запуск NetBeans

Вы должны установить Java 7.0 до установки NetBeans. Если вы устанавливали JDK с веб-сайта Sun, возможно, вы загрузили совмещенный пакет JDK и NetBeans, так что JDK уже установлена у вас. Отдельный пакет установки NetBeans вы можете найти на сайте **www.netbeans.org**. Следуйте простым инструкциям программы установки этого пакета (возможно, вам потребуется перезагрузка, если вы устанавливаете Java). По завершении, запустите NetBeans. При первом запуске NetBeans появится окно приветствия. Пока закройте его, нажав на небольшой крестик на его панели. Если вы хотите повторно вызвать окно приветствия, чтобы получить консультацию по NetBeans, выберите меню **Help ⇒ Welcome Screen** (Справка ⇒ Начальная страница).

Установка bean-компонентов

Перед началом работы вы должны добавить bean-компоненты, используемые в этой главе, на панель **Palette** (Палитра) в окне NetBeans. NetBeans устанавливает компоненты Java beanв расположение, которое может использоваться несколькими проектами в редакторе.

- Для установки bean-компонентов, загрузите JAR-файл *magicbeans.jar* с веб-сайта этой книги **(eksmo.ru/smv/Primers.zip)**.
- Сохраните файл на жестком диске и выберите команду меню **Tools ⇒ Palette ⇒ Swing/AWT Components** (Сервис ⇒ Палитра ⇒ Компоненты Swing/AWT) для добавления bean-компонентов.
- Нажмите кнопку **Add from Jar** (Добавить из архива JAR), выберите файл *magicbeans.jar* и нажмите кнопку **Next** (Далее). Мастер отобразит список bean-компонентов, содержащихся в загруженном JAR-файле (их должно быть восемь)
- Выберите их все (выберите первый и, нажав и удерживая клавишу **Shift**, щелкните мышью по последнему), затем нажмите кнопку **Next** (Далее).

- NetBeans предложит выбрать желаемую категорию панели **Palette** (Палитра) для размещения файла. Выберите вариант **Beans** (Компоненты), а затем нажмите кнопки **Finish** (Готово) и **Close** (Закрыть).

Мы увидим эти bean-компоненты, как только начнем редактировать приложение.

Создание проекта

Теперь мы должны создать новый проект. Для создания нового проекта выберите пункт **New Project** (Создать проект) в меню **File** (Файл). В области **Categories** (Категории) выберите пункт **Java**, а в области **Projects** (Проекты) — вариант **Java Application** (Приложение Java). Затем нажмите кнопку **Next** (Далее). Укажите название вашего проекта, например «LearningJava», и выберите директорию или оставьте указанную по умолчанию. Сбросьте флажок **Create Main Class** (Создать главный класс), так как мы создадим собственный, а затем нажмите кнопку **Finish** (Готово) для создания пустого проекта. Если вы создаете проект в NetBeans впервые, то вы можете увидеть сообщение о том, что NetBeans сканирует директорию classpath. Это может занять несколько минут.

Далее нам необходимо создать файл класса для добавления в проект. Выберите пункт **New File** (Создать файл) в меню **File** (Файл). В области **Categories** (Категории) выберите пункт **Swing GUI Forms** (Формы Swing GUI), а в области **File Types** (Типы файлов) — вариант **JFrame Form** (Форма JFrame). Выбор последнего пункта предоставляет нам Java-класс, дополняя `JFrame` основной структурой уже настроенного GUI приложения. Нажмите кнопку Далее, укажите имя файла — LearningJava1. Если желаете, то можете оставить набор пакетов по умолчанию. Теперь нажмите кнопку **Finish** (Готово); экран должен выглядеть так, как показано на рис. 22.1.

Рабочая область NetBeans

На рис. 22.1 изображено приложение NetBeans. Экран состоит из трех главных областей, с панелью инструментов в верхней части. Область в левой части — это окно проводника, отображающее Java проект, файлы и службы на выбор. Проводник в нашем случае отображает наш класс. В левой нижней части расположена область **Navigator** (Навигатор), в которой представлены методы и свойства класса и, в случае, с GUI-компонентом, он отображается вместе с дочерним компонентом. В центральной части представлена рабочая область с вкладками

для каждого файла (в данный момент одного). Так как мы выбрали тип файла GUI, NetBeans примет интерфейс для разработки GUI и покажет нам пустой прямоугольник, в котором мы будем размещать наши bean-компоненты. Попробуйте переключиться между текущим представлением и исходным кодом Java, выбрав кнопки **Source** (Источник) или **Design** (Проект) соответственно.

Панели **Palette** (Палитра) и **Properties** (Свойства), размещенные в правой части окна, сейчас отражают некоторые свойства выбранного файла, но когда мы добавим bean-компонент в область проектирования, будут отражать его свойства. Панель **Palette** (Палитра) группирует различные типы bean-компонентов. Группы **Swing** включают все стандартные Swing-компоненты. Группа **AWT** содержит более старые компоненты AWT. Группа **Layouts** (Разметки) содержит инструменты управления разметкой Java.

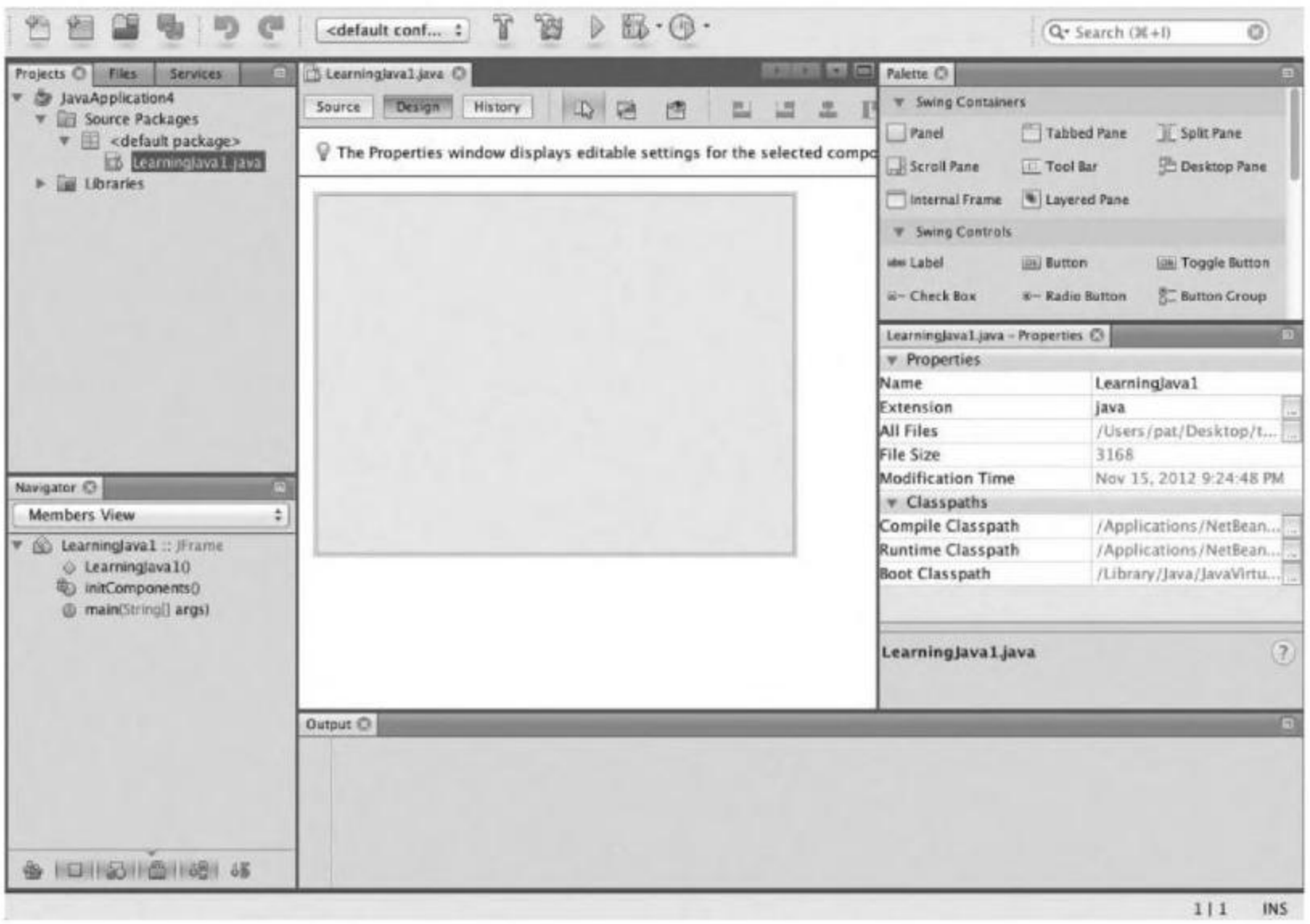

Рис. 22.1. Рабочая область NetBeans

В панели **Palette** (Палитра) выберите группы с именем **Swing** (они могут быть раскрыты по умолчанию) для просмотра доступных Swing-компонентов (они отображаются как значки в верхней части окна). Теперь прокрутите содержимое панели вниз и раскройте группу **Beans** (Компоненты), чтобы увидеть bean-компоненты, импортированные

нами ранее. Вы должны увидеть bean-компонент `Dial` из главы 18, вместе с bean-компонентом крошечного человечка. Оставшиеся bean-компоненты не оформлены симпатичными значками потому, что не были упакованы вместе с примерами. Позже в этой главе мы расскажем о создании пакетов. На рис. 22.2 показана группа **Beans** (Компоненты) панели **Palette** (Палитра).

Чтобы разместить bean-компонент в рабочей области, щелкните по нему мышью, а затем щелкните мышью в рабочей области. Перед тем как мы выполните это, возможно, вы захотите настроить соответствующее управление разметкой. Панель **Navigator** (Навигатор) содержит дерево, отражающее все компоненты проекта (скрытые и нет). Щелкнув правой кнопкой мыши по `JFrame` (контейнеру верхнего уровня) в рабочей области или в дереве, вы можете выбрать команду **Set Layout** (Установить расположение), чтобы определить управление разметкой для данного фрейма. На данный момент попробуйте использовать **Absolute Layout** (Абсолютное форматирование), предоставляемое NetBeans. Так вы сможете свободно позиционировать и перемещать bean-компонент в контейнере. Обычно этого не требуется, но в нашем случае пример станет проще понять. Другие способы управления разметкой достаточно просты, и вы можете обратиться к руководству пользователя NetBeans за подробностями.



Рис. 22.2. Группа **Beans** панели **Palette**

Свойства и настройки

Свойства представляют «состояние» или «данные» содержимого bean-компонента. Этими функциями можно управлять для конфигурации bean-компонента. Для bean-компонента, который является GUI, как

вы могли догадаться, доступны такие настройки, как размер, цвет и другие характеристики его основного представления. В принципе, свойства похожи на открытые переменные объектов. Как и переменные, свойства могут быть примитивного типа (число или булево значение) или сложного объектного типа (как `String` или коллекция данных электронных таблиц). В отличие от переменных, свойства всегда управляются методами, устанавливающими и получающими их значения, что позволяет bean-компоненту реагировать на каждое изменение свойства. Отправляя событие при изменении свойства, bean-компонент может уведомить другие заинтересованные bean-компоненты об изменении (каким образом — мы обсудим позже в этой главе).

Давайте добавим пару bean-компонентов в NetBeans и рассмотрим их свойства. Нажмите кнопку **Button** (Кнопка) (это будет `JButton`) в группе **Swing Controls** (Элементы управления Swing) на панели **Palette** (Палитра) и затем щелкните мышью в рабочей области. Когда компонент `JButton` будет впервые загружен в NetBeans, свойства будут проанализированы. Когда мы выберем экземпляр кнопки, NetBeans отобразит его свойства на панели **Properties** (Свойства), и вы сможете изменять их.

Кнопка имеет восемь основных свойств в нескольких группах. Группа **Other Properties** (Другие свойства) содержит дополнительные свойства. Ниже расположены группы **Layout** (Макет) (на самом деле, `JButton` не располагает настраиваемыми свойствами в этой группе) и **Accessibility** (Доступность) (параметры доступности компонента). Свойства `foreground` и `background` — это цвета; их текущие значения отражаются в соответствующих блоках. Свойство `font` определяет шрифт текстовой надписи; пример шрифта приведен рядом. Свойство `text` определяет текст надписи на кнопке.

Вы также можете выбрать изображение для кнопки, указать текст подсказки, которая появляется при наведении указателя мыши на элемент и идентификатор сочетания клавиш, называемый *мнемоникой*. Измените значение свойства `font` и просмотрите результат настройки надписи кнопки. Щелкните мышью по свойству `background` для ввода численного значения цвета или нажмите кнопку **...**, чтобы вызвать всплывающее диалоговое окно выбора цвета.

Большинство основных свойств будут вам знакомы, так как многие GUI bean-компоненты наследуют свои свойства от основного класса `JComponent`. Группа **Other Properties** (Другие свойства) содержит спи-

сок из почти 50 дополнительных свойств, наследуемых от `JComponent`. NetBeans делает попытку категоризировать их. Как мы увидим во время создания собственного bean-компонента, мы можем ограничить свойства, которые отображаются на панели **Properties** (Свойства).

Теперь поместите bean-компонент `Juggler` в рабочую область (это один из предоставленных компанией Sun компонентов Java bean, который мы обновили). Запустится анимация, и жонглер начинает жонглировать несколькими схематично нарисованными бобами, как показано на рис. 22.3. Если это вас раздражает, не беспокойтесь: скоро мы возьмем их под контроль.

Как видите, этот bean-компонент имеет другой набор свойств, одно из которых, названное `animationRate`, вызывает интерес. Это целочисленное свойство, которое контролирует задержку между отображением кадров жонглера в миллисекундах.

Попробуйте изменить его значение. Скорость жонглирования меняется каждый раз, когда вы вводите значение. Хороший bean-компонент незамедлительно реагирует на изменение его свойств. Сбросьте флажок со свойства `juggling`. Анимация будет остановлена.

Заметим, что панель **Properties** (Свойства) предоставляет способ отображения и редактирования каждого из различных типов свойств. Для свойств `foreground` и `background` компонента `JButton` панель показывает цвета; если вы щелкнете по ним мышью, появится всплывающее диалоговое окно выбора цвета.

Аналогично, если вы щелкните мышью по свойству `font`, откроется диалоговое окно выбора шрифта. При вводе значений `integer` и `string` вы можете указывать их непосредственно в соответствующих полях. NetBeans понимает и может редактировать большинство полезных основных свойств Java.

Некоторые bean-компоненты, имеющие свойства более сложных типов, могут предлагать собственные *редакторы свойств*. Bean-компонент `Molecule`, с которым мы поэкспериментируем в следующей части, использует собственный редактор свойств, позволяющий выбирать тип молекулы. Если необходим еще больший контроль над отображением свойств, bean-компонент может предоставить *настройщики*, позволяющие bean-компонентам реализовывать их собственный GUI для редактирования собственных свойств (например, настройщик мог бы позволить вам нарисовать значок для кнопки).

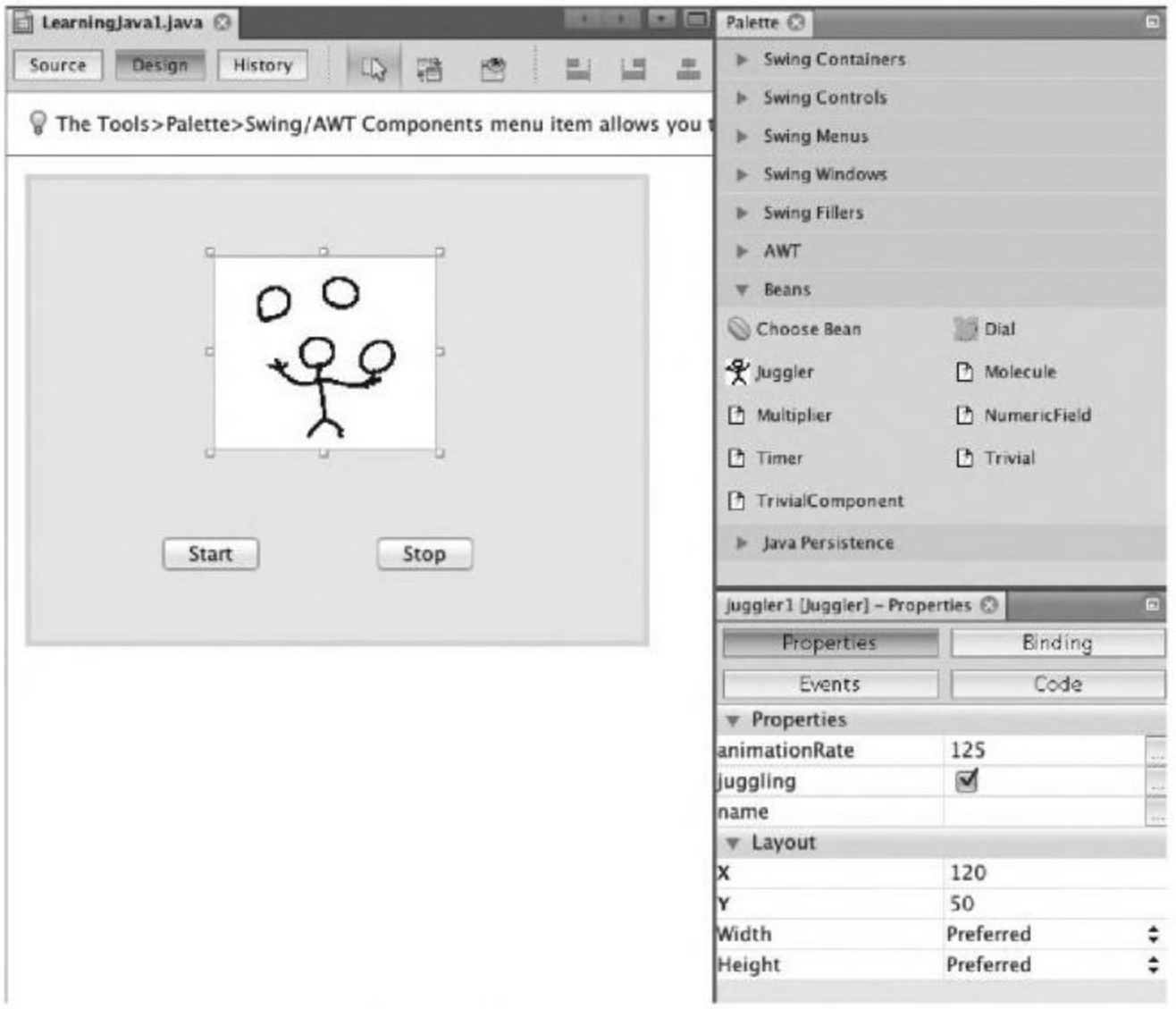

Рис. 22.3. Жонглирование бобами

Подключения и адаптеры событий

Bean-компоненты используют события для взаимодействия. Как мы упоминали в главе 16, события не ограничены GUI-компонентами и могут использоваться для оповещения и передачи информации в более главное приложение. Событие — это лишь уведомление; информация, описывающая событие и другие данные, вложена в подкласс `EventObjects` и передается принимающему объекту через вызов метода. Событие-источник регистрирует прослушиватель, необходимый для получения события при его наступлении. Событие-получатель реализует соответствующий интерфейс прослушивателя, содержащий методы необходимые для получения событий. Это краткое представление механизма Java.

Часто бывает полезно поместить адаптер между событием-источником и прослушивателем. Адаптер может использоваться, когда объект не знает, как получить конкретное событие; он включает обработку события объектом любым способом. Адаптер может преобразовывать событие в некоторое другое действие, такое как вызов другого метода или обновление некоторых данных. Одна из задач NetBeans — помочь подключить событие-источник в прослушивателе события. Другая за-

дача — предоставлять код адаптера, который позволяет нам подключать события более сложными способами.

Добавление таймера в приложение `Juggler`

Возьмем нашего жонглера под контроль, используя следующие шаги:

- Используя панель **Properties** (Свойства), измените надпись кнопки на «Start».
- Теперь нажмите кнопку **Connection Mode** (Режим соединения) в верхней части вкладки **Design** (Проект) (второй значок, изображающий два окна со стрелками, направленными в противоположные стороны)
- После нажатия кнопки NetBeans ожидает от нас выбора двух компонентов для подключения. Сначала щелкните мышью по кнопке `Start`, а затем — по кнопке `Juggler`. NetBeans откроет диалоговое окно **Connection Wizard** (Мастер соединения), указывающее компонент-источник (кнопку) и отображающий список событий (см. рис. 22.4) Большинство из них — стандартные события Swing, которые могут быть созданы любым типом класса `JComponents`. Позже мы используем его как событие действия кнопки.
- Разверните папку с именем *action* и выберите пункт `actionPerformed` в качестве события источника.
- В нижней части диалогового окна NetBeans указано имя метода обработчика события, который будет автоматически создан. Оставьте имя метода установленным по умолчанию. Нажмите кнопку **Next** (Далее), чтобы перейти к следующему окну, предлагающему указать операцию цели для компонента `Juggler`.
- Вы можете выбрать свойство для настройки `Juggler`, как показано на рис. 22.5. Показано три свойства `Juggler`. Выберите пункт `juggling` в качестве цели и нажмите кнопку **Next** (Далее).
- В следующем окне введите значение **`true`** в поле **Value** (Значение) и нажмите кнопку **Finish** (Готово). NetBeans отобразит исходный код и выделит метод, созданный для реакции на действия кнопки.

Мы завершили соединение между кнопкой и `Juggler`. Когда кнопка вызывает действие события, свойству `juggling` компонента `Juggler` присваивается значение `true`.

Просмотрите исходный код и обратите внимание на фрагмент, который сгенерировал NetBeans, чтобы создать это соединение. Конкретно, в методе `initComponents()` нашего класса был создан анонимный внутренний класс, служащий как `ActionListener` для `ActionEvents`, поступающих от нашей кнопки (которая называется `jButton1`):

```
jButton1.addActionListener(new java.awt.event.ActionListener() {
   public void actionPerformed(java.awt.event.ActionEvent evt) {
      jButton1ActionPerformed(evt);
   }
});
```

Адаптер вызывает частный метод, который устанавливает свойства нашего `Juggler`:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
   Juggler1.setJuggling(true);
}
```

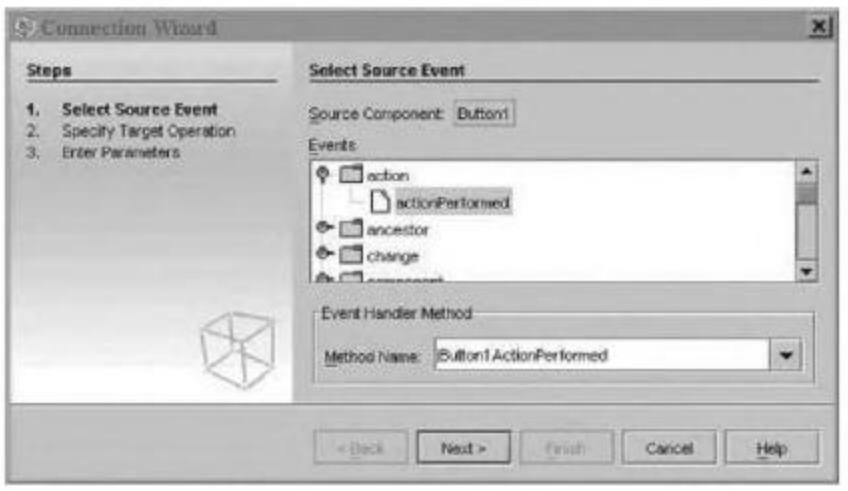

Рис. 22.4. Выбор события источника в окне **Connection Wizard**

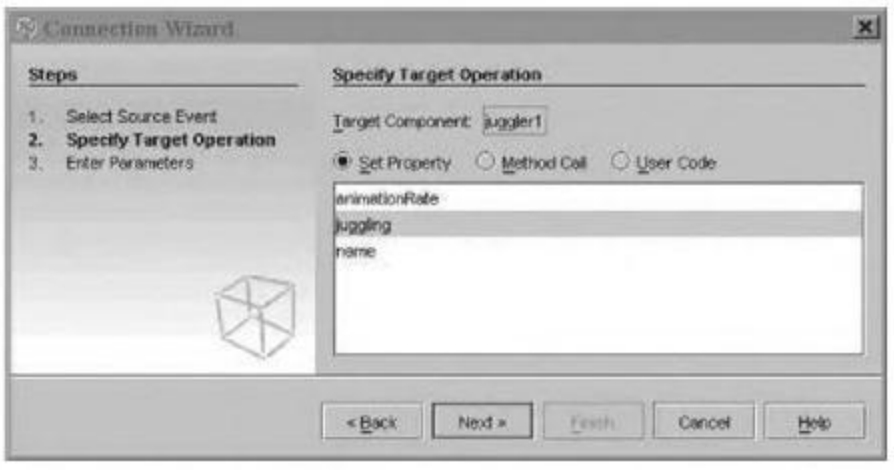

Рис. 22.5. Определение цели операции в окне **Connection Wizard**

Вы заметите, что бóльшая часть кода, созданного автоматически, выделена серым цветом, указывающим, что его невозможно изменить вручную. Но тело частного метода открыто, и мы можем произволь-

но изменить его код, чтобы настроить действие при нажатии кнопки. В NetBeans соединение — это только начальная точка.

Все это может показаться немного непонятным. В конце концов, если мы поставим на первое место создание компонента `Juggler` через `ActionListener`, мы должны ожидать связи напрямую с кнопкой. Однако, как мы увидим позже, использование адаптеров неразрывно связано с гибкостью.

Чтобы завершить наш пример, перейдите на вкладку **Design** (Проект), затем повторите весь процесс, добавив кнопку с надписью «Стоп». Мы могли бы использовать для ее создания тот же способ, что и для кнопки «Старт»: передавая указанное значение в метод `juggling`, но попробуем другой вариант. Нажмите кнопку **Connection Mode** (Режим соединения); щелкните мышью по кнопке «Stop» и `Juggler`. Также выберите метод `actionPerformed` как источник, но в этот раз вместо выбора свойства компонента Juggler установите переключатель в положение **Method Call** (Вызов метода), чтобы увидеть список методов, доступных для bean-компонента `Juggler`. Прокрутите список до конца и выберите метод **`stopJuggling()`**. Нажмите кнопку **Finish** (Готово), чтобы завершить соединение, и взгляните на созданный код. Так мы можем видеть пример установки произвольного метода, вызываемого bean-компонентом. Разумеется, так же можно выбрать и метод `startJuggling()` для первой кнопки.

Выполнение примера

Теперь соберем `Juggler`. Щелкните правой кнопкой мыши по файлу *LearninJava1.java* на вкладке **Projects** (Проекты) панели **Explorer** (Проводник) и выберите команду **Run File** (Запустить файл) в контекстном меню. Подождите, пока NetBeans компилирует и запускает наш пример. Теперь мы можем остановить и запустить жонглера, используя кнопки! Наигравшись, закройте приложение `juggler` и вернитесь в редактор. Завершите работу с примером, закрыв вкладку в рабочей области. (Нет необходимости вручную сохранять файл; NetBeans автоматически сохранит внесенные изменения.)

Движение молекул

Рассмотрим еще один интересный пример, показанный на рис. 22.6. Создайте новый файл в проекте, как делали это ранее, выбрав пункт **New**

File (Создать файл) в меню **File** (Файл). В области **Categories** (Категории) выберите пункт **Swing GUI Forms** (Формы Swing GUI), а в области **File Types** (Типы файлов) — вариант **JFrame Form** (Форма JFrame). Присвойте файлу имя «LearningJava2».

Перетащите с панели **Palette** (Палитра) bean-компонент `Molecule` и разместите его в рабочей области. Теперь, если вы запустите пример, вы сможете вращать модель в трех направлениях, перетаскивая ее мышью. Попробуйте изменить молекулу, используя панель **Properties** (Свойства)*.

Посмотрим, что мы можем сделать с нашей молекулой. Используем bean-компонент `Timer`, доступный на панели **Palette** (Палитра)**. Каждую указанную миллисекунду `Timer` вызывает событие. `Timer` контролируется свойством `delay` типа `long`, которое определяет количество миллисекунд между событиями. `Timer` — один из «скрытых» bean-компонентов; он не относится к классу `JComponent` и не имеет графического представления, также как внутренний таймер приложения обычно не присутствует на экране. Скрытые bean-компоненты перечислены на панели **Navigator** (Навигатор) в левой части окна NetBeans, как любые другие bean-компоненты. Когда необходимо выбрать bean-компонент `Timer`, щелкните по нему мышью на панели **Navigator** (Навигатор).

Объединим компоненты `Timer` и `Molecule`. Нажмите кнопку **Connection Mode** (Режим соединения) и последовательно выберите компоненты `Timer` и `Molecule`. Выберите событие `timerFired()` компонента `Timer` из списка. Нажмите кнопку **Next** (Далее) и установите переключатель в положение **Method Call** (Вызов метода). Выберите метод **`rotateOnX()`** и нажмите кнопку **Finish** (Готово). Запустите пример. Теперь компонент `Molecule` должен вращаться вокруг своей оси каждый раз при получении события от `Timer`. Попробуем изменить значение свойства `delay`. Вы можете передать от компонента `timer` компоненту `Molecule` метод `rotateOnY()`. Используйте различные экземпляры `Timer` и, устанавливая в них различные значения `delay`, заставьте его вращаться с разной скоростью и в разных направлениях.

---

* На момент написания книги в примере Sun Molecule есть некоторые проблемы при использовании NetBeans. Выбор отличной от предустановленной молекулы приводит к ошибке компиляции. Вы можете использовать кнопку **Preview Design** (Просмотр проекта) в редакторе NetBeans , чтобы испытать другие молекулы.

** Bean-компонент `Timer` поставлялся вместе с пакетом установки NetBeans, но исчез в версии 4.0, поэтому мы заменили его собственным. Мы не будем обсуждать его здесь, в его коде нет ничего, с чем мы не ознакомимся, рассматривая другие примеры.

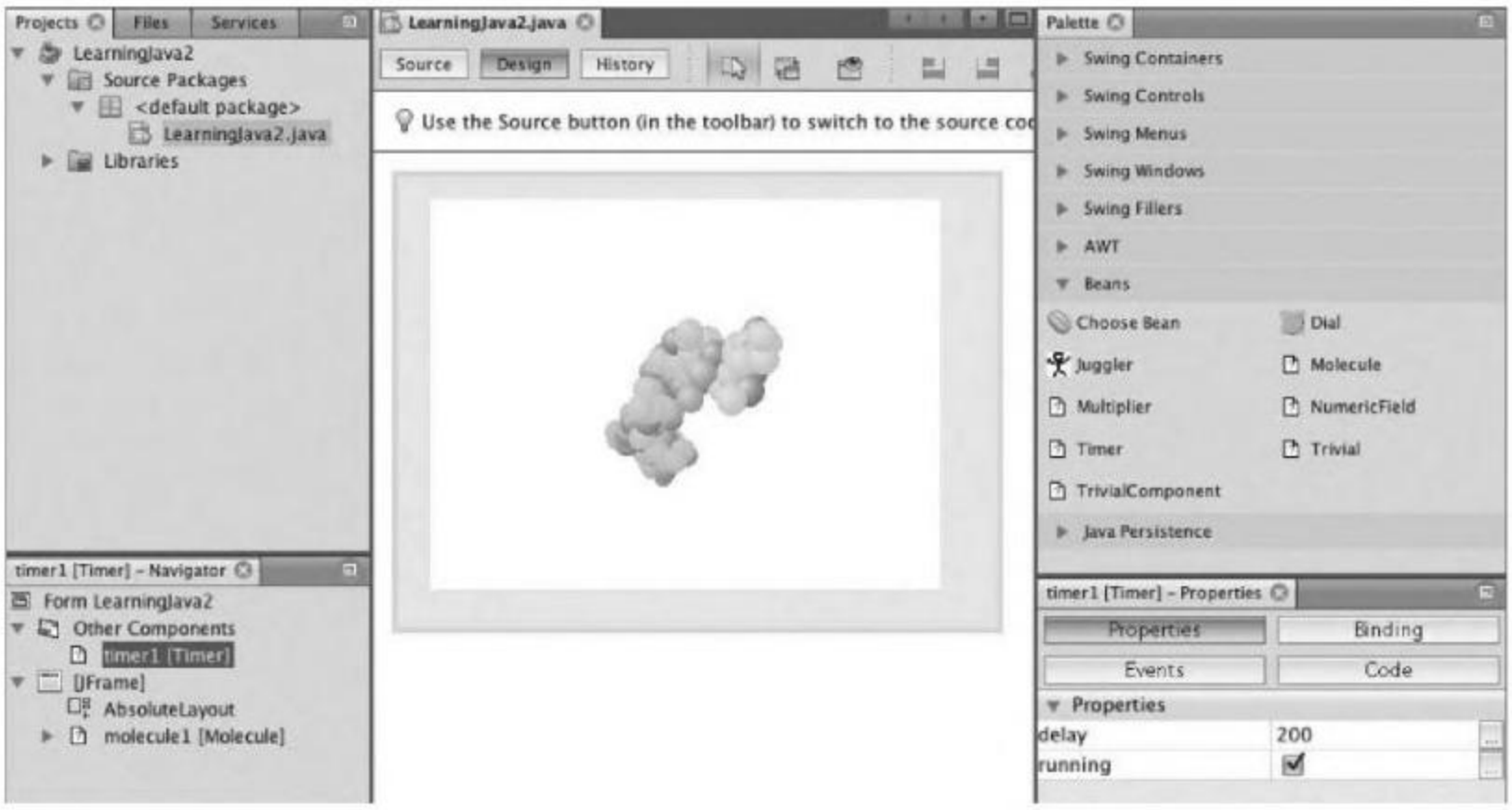

Рис. 22.6. Bean-компоненты `Molecule` и `Timer`

Связывание свойств

Используя комбинации событий и адаптеров, мы можем соединять bean-компоненты множеством интересных способов. Мы можем даже «связать» два bean-компонента вместе так, что, если свойство изменяется в первом bean-компоненте, соответствующее свойство автоматически изменяется во втором. При таком сценарии bean-компоненты не обязательно должны быть одного типа, чего нельзя сказать о свойствах.

Закроем файл `Molecule` и создадим еще один. Перетащим два bean-компонента `NumericField` из панели **Palette** (Палитра) на рабочую область и выберем один из них. Возможно, вы вновь захотите использовать **Absolute Layout** (Абсолютное форматирование). Вы также можете регулировать ширину полей, перетащив их мышью. Вы заметите, что `NumericField` имеет множество стандартных свойств Swing-компонента. Если вы взглянете на группу **Other Properties** (Другие свойства) на панели **Properties** (Свойства), то увидите свойство `value`, которое представляет численное значение поля. Вы можете установить его здесь или ввести число в поле при запуске программы. `NumericField` игнорирует буквенные символы.

Давайте свяжем свойства `value` двух полей. Нажмите кнопку **Connection Mode** (Режим соединения) для создания соединения между двумя полями. Щелкните мышью по компоненту `numericField1`, а затем — по компоненту `numericField2`. Таким образом `numericField1`

станет источником. В окне **Connection Wizard** (Мастер соединения) выберите `propertyChange()` — событие поля источника. Это метод для `PropertyChangeEvent` — созданное событие отправляется bean-компонентами при изменении одного из их свойств. Когда вызванное bean-компонентом свойство изменяет событие в соответствии с конкретным свойством, это свойство называется «связанным». Это значит, что можно связать свойства других классов используя основной механизм. В этом случае свойство `value` наших bean-компонентов `NumericField` — это связанные свойства, так что при их изменении запустится `PropertyChangeEvent`.

Нажмите кнопку **Next** (Далее) и укажите значение `value`, как цель `numericFieldv2`. Еще раз нажмите кнопку **Next** (Далее) и установите переключатель в положение **Properties** (Свойства). Нажмите кнопку **...**, чтобы открыть диалоговое окно **Select Property** (Выбор свойства). Выберите числовое поле (`numericField1`, если это ваша кнопка-источник) в списке, а затем выберите свойство **`value`**. Нажмите последовательно кнопки **OK** и **Finish** (Готово) для завершения соединения.

Запустите приложение и попробуйте ввести некое значение в первое поле `numericField1`. Второе поле должно измениться одновременно. Значение свойства второго bean-компонента будет связано с первым.

Попробуем связать значения свойств в другом направлении так, чтобы вы могли изменять значение в обоих направлениях. Простая логика в bean-компоненте прекратит бесконечный цикл, происходящий в данном случае.

NetBeans вновь автоматически создаст адаптер. В этот раз он прослушивает `PropertyChangeEvents` и вызывает метод `setValue()` нашему целевому полю. Мы не делаем ничего сверхъестественного. `PropertyChangeEvents` запрашивает некоторые дополнительные данные — старое и новое значения свойства, но мы не используем их здесь. Используя окно **Connection Wizard** (Мастер соединения), вы можете применять любые события-источники. Наконец, мы видим, что свойства могут передавать значения от одного bean-компонента другому в текущей разметке. Если вы заинтересованы в определении свойств, которые изменены, или, если вы хотите применить логику значений, можете дополнить код созданного метода собственным.

Многие Swing-компоненты имеют связанные свойства, которые обычно документируются в Javadoc-данных классов.

Принудительные свойства

В предыдущей части мы говорили о том, как bean-компоненты вызывают `PropertyChangeEvents` для уведомления других bean-компонентов (и адаптеров) о том, какое свойство было изменено. JavaBeans также поддерживает *составные свойства*, в которых прослушиватели событий сообщают, существует ли доступный bean-компонент для изменения значения свойства. Если новое значение не принимается, изменения отменяются; событие-источник сохраняет свое старое значение.

Концепция сдерживания свойств не используется активно в обычных операциях Swing, поэтому мы не будем ее подробно рассматривать. Но происходит приблизительно следующее. Обычно `PropertyChangeEvents` передается методу `propertyChange()` в прослушивателе. Составные свойства реализуются передачей `PropertyChangeEvents` отдельному методу прослушивателя, называемому `vetoableChange()`. Этот метод выполняется через `PropertyVetoException`, если это не предполагаемое изменение. В этом случае, компоненты могут управлять доступными значениями, установленными другими компонентами.

Создание bean-компонентов

Теперь, когда вы знаете, как выглядят bean-компоненты с точки зрения пользователя, приступим к их созданию. Мы создадим кое-какие bean-компоненты, упакуем их и используем NetBeans для создания простого приложения. Готовый JAR-файл и исходный код примеров этой главы доступны по адресу **eksmo.ru/smv/Primers.zip**.

Первое, о чем мы напомним: абсолютно все может быть bean-компонентом. Даже следующий класс — это bean-компонент, хотя и скрытый:

```
public class Trivial implements java.io.Serializable{}
```

Конечно, такой bean-компонент не слишком полезен: он не имеет никаких свойств и ничего не делает. Но, тем не менее, это bean-компонент, и мы можем переместить его в NetBeans, как только корректно упакуем. Если мы изменим этот класс на расширяемый `JComponents`, мы получим графический bean-компонент, который будет виден на разметке, со многими стандартными свойствами Swing-компонентов, такими, как информация о цвете и размере:

```
public class TrivialComponent extends JComponent{}
```

Bean-компонент `Dial`

Мы создали изящный компонент `Dial` в главе 18. Что необходимо сделать, чтобы превратить его в bean-компонент? Сюрприз: это уже bean-компонент! `Dial` имеет набор свойств, который он раскрывает способом, предписанным JavaBeans. Геттер получает значение свойства: например, `getValue()` возвращает текущее значение `Dial`. Аналогично, метод `set`, `SetValue()`, изменяет значение `Dial`. `Dial` имеет два других свойства, которые также имеют геттеры и сеттеры: `minimum` и `maximum`. Это все, что необходимо `Dial`, чтобы передать информацию такому инструменту, как NetBeans, о том, какие свойства он имеет и как с ними работать. Так как `Dial` — это `JComponent`, он также имеет все стандартные свойства Swing-компонентов, например цвет и размер. Класс `JComponent` предоставляет геттеры и сеттеры для всех свойств.

Чтобы использовать наш компонент `Dial`, мы поместим его в Java-пакет `magicbeans` и сохраним его в JAR-файл, который можно будет загрузить в NetBeans. Исходный код, который можно найти на сайте **eksmo.ru/smv/Primers.zip**, включает Ant build файл (см. главу 15), который компилирует код и создает итоговый JAR-файл.

Если вы хотите начать работу, это должно выглядеть так: во-первых, создайте директорию *magicbeans* для хранения нашего bean-компонента, добавьте выражение `package` в исходные файлы *Dial.java*, *DialEvent.java* и *DialListener.java* (см. главу 18), поместите исходные файлы в директорию *magicbeans* и скомпилируйте их (используя команду `javac magicBeans/Dial.java`) для создания файлов класса.

Далее нам необходимо создать файл декларации, который сообщит NetBeans, какие из классов в JAR-файле являются bean-компонентами и какие файлы являются поддерживаемыми или не требуются. К этому моменту у нас имеется только один bean-компонент, *Dial.class*, поэтому создадим следующий файл с именем *magicbeans.manifest*:

```
Name: magicbeans/Dial.class
Java-Bean: True
```

`Name` определяет как будет описываться файл внутри JAR: `magicbeans/Dial.class`. Все технические характеристики описанные между строками `Name:` и следующей пустой строкой относятся к элементу, указанному в строке `Name`. Мы добавили атрибут `Java-Bean: True`, который помечает данный класс, как bean-компонент, для инструментов, считывающих данный манифест. Мы добавили такие же записи для каждого bean-ком-

понента нашего пакета. Нам нет необходимости отмечать сопровождающие классы (такие, как `DialEvent` и `DialListener`), как bean-компоненты. Так как фактически мы не хотим управлять ими непосредственно через NetBeans, мы не должны их упоминать в манифесте вовсе.

Чтобы создать JAR-файл, содержащий данные манифеста, используйте команду:

```
% jar -cvmf magicbeans.manifest magicbeans.jar magicbeans/*.class
```

Если вы загрузили скомпилированные примеры, как описывалось раньше, тогда у вас уже есть bean-компонент `Dial`, загруженный в NetBeans. В версии, доступной в скомпилированном файле *magicbeans.jar*, имеется дополнительный упаковщик, который позволяет ей отображаться с великолепным значком на панели **Palette** (Палитра), о чем мы поговорим немного позже. (Если вы еще не загрузили JAR-файл примера, вы можете импортировать тот, который мы только что создали, вызвав диалоговое окно **Palette → Swing/AWT Components** (Палитра → Компоненты Swing/AWT) из меню **Tools** (инструменты), как было ранее описано в этой главе.) Если вы желаете заменить существующий bean-компонент `Dial` на вашей панели **Palette** (Палитра), вы можете удалить его, выбрав пункт локального меню изображения **Delete** (Удалить) прежде, чем выполнить импорт JAR-файла. (На самом деле, NetBeans 7.2 должен перезагружать JAR-файлы автоматически, если вы перезаписываете их.)

Теперь на вашей панели **Palette** (Палитра) должен появится bean-компонент `Dial`. Поместите экземпляр bean-компонента `Dial` в новый файл JFrame Form в NetBeans.

Как показано на рис. 22.7, свойства `Dial` (`maximum`, `minimum`, и `value`) доступны на панели **Properties** (Свойства) и могут быть изменены в NetBeans. Если вы создали JAR-файл `Dial`, следуя нашим кратким инструкциям, вы увидите эти свойства вместе со всеми свойствами Swing, наследованными от класса `JComponent`. Bean-компонент `Dial` представлен на рисунке в том виде, в котором он будет показан позже в этой главе (с помощью прилагаемого упаковщика), после того, как мы познакомимся с классом `BeanInfo`. Мы почти у цели.

Сейчас мы готовы к использованию `Dial`. Вновь откройте файл `Juggler`, который мы создали в первой части этой главы. Добавьте экземпляр нашего нового волшебного bean-компонента `Dial` в сценарий, как показано на рис. 22.8.

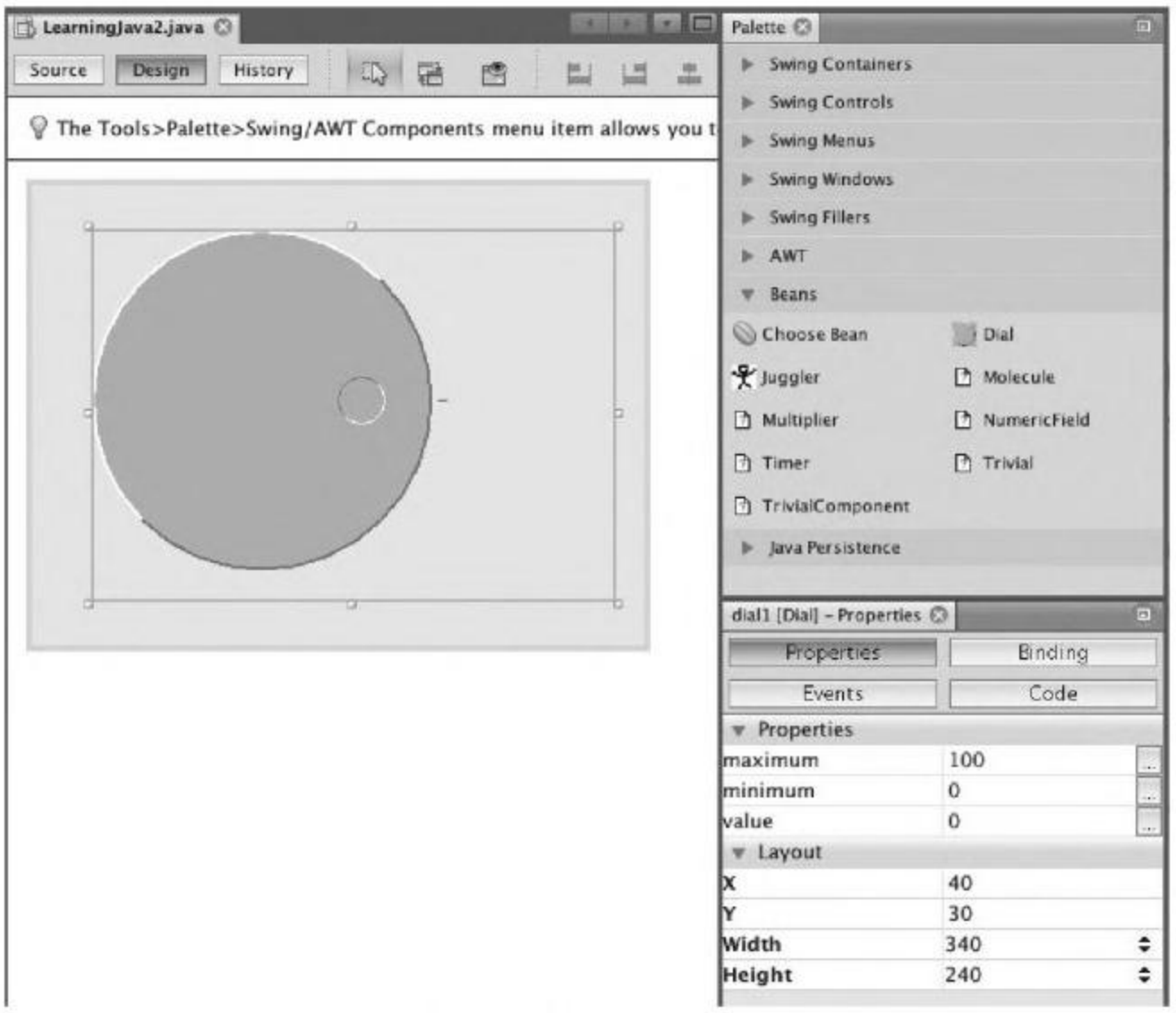

Рис. 22.7. Bean-компонент `Dial`

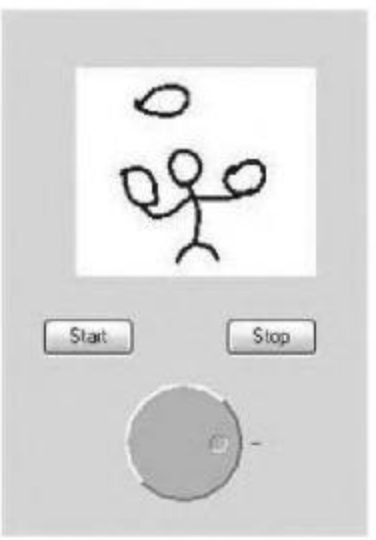

Рис. 22.8. `Juggler` с регулятором скорости анимации.

Свяжем свойство `value` компонента `Dial` со свойством `animationRate` компонента `Juggler`. Используйте диалоговое окно **Connection Wizard** (Мастер соединения), как ранее, выберите последовательно компоненты `Dial` и `Juggler`.

Выберите источник `DialEvent` и соедините свойство `animationRate`, выбрав свойство `value` в качестве источника. Когда вы завершите перехват, вы сможете изменять скорость жонглера с помощью ручки регулятора. Попробуйте изменить значения свойств `maximum` и `minimum` компонента `dial` для настройки диапазона.

Разработка шаблонов для свойств

Ранее мы говорили, что такие инструменты, как NetBeans, получают информацию о свойствах bean-компонентов, используя геттеры и сеттеры. Простейший способ сделать свойство видимым — это придерживаться следующих простых шаблонов проектирования:

- Метод для получения значения текущего свойства:

```
public PropertyType getPropertyName()
```

- Метод для установки значения свойства:

```
public void setPropertyName( PropertyType arg )
```

- Метод, определяющий равно ли `true` текущее значение свойства булевого типа:

```
public boolean isPropertyName()
```

Последний метод дополнительный и используется только для свойств со значениями типа Boolean. (Вы можете использовать геттер для значений типа Boolean, чтобы получить тот же результат.)

Соответствующие геттеры и сеттеры для функций нашего bean-компонента, добавленные нами или наследованные от классов `java.awt.Component` и `javax.swing.JComponent`, присутствуют в классе `Dial`:

```
// наследование от Component
public Color getForeground()
public void setForeground(Color c)
public Color getBackground()
public void setBackground(Color c)
public Font getFont()
public void setFont(Font f)
// множество других из Component и JComponent
// собственная часть Dial
public int getValue()
public void setValue(int v)
public int getMinimum()
public void setMinimum(int m)
public int getMaximum()
public void setMaximum(int m)
```

JavaBeans допускает ограничение возможности чтения и записи свойств, что реализуется простым исключением одного из геттеров или

сеттеров. NetBeans использует API отражения для получения информации о методах bean-компонента `Dial`; впоследствии используя это соглашение имен, чтобы узнать, какие из свойств доступны. Когда мы используем редактор свойств для изменения значения, NetBeans динамически вызывает верный сеттер для изменения значения.

Рассмотрев подробнее класс `JComponent`, вы заметите, что другие методы придерживаются указанного шаблона проектирования. Например, что можно сказать о паре методов `setCursor()` и `getCursor()`? NetBeans не знает, каким образом отображать или редактировать курсор, а мы не предоставили соответствующий редактор, поэтому эти свойства не отражаются на странице свойств.

NetBeans автоматически извлекает имя свойства из имени его метода доступа; далее имя преобразуется в нижний регистр для отображения на странице свойств. Например, свойство `font` получено из метода `getFont`. Позже мы увидим, как создать класс `BeanInfo`, переопределяющий способ отображения свойств, что позволит вам использовать собственные понятные имена свойств. Опять же, если вы используете bean-компонент `Dial` из нашего предварительно скомпилированного примера JAR, вы увидите только три свойства `Dial`. Свойства класса `JComponent` скрыты в нашем пакете с помощью класса `BeanInfo`, где определены свойства, которые мы хотим отобразить.

Создание шаблонов bean-компонентов в NetBeans

NetBeans автоматически распознает JavaBeans-шаблоны геттеров и сеттеров. В исходном коде, в меню **Source** (Источник) выберите пункт **Insert Code** (Вставка кода) и подпункт **Getter and Setter** (Методы Get и Set) для автоматического создания геттеров и сеттеров для полей вашего класса. Это поможет сократить объем набираемого кода, если вам необходимо добавить много свойств.

Ограничения визуальной разработки

Эти примеры демонстрируют, что мы можем создать простое приложение, связав вместе bean-компоненты, большей частью визуально. В других средах разработки, такой вид связывания bean-компонентов развит еще лучше. Например, оригинальный «BeanBox» компании Sun — экспериментальный контейнер компонентов Java bean — использует иной подход, нежели NetBeans. Он позволяет разработчикам рабо-

тать с экземплярами bean-компонентов в режиме реального времени, динамически создавая код адаптера во время выполнения и полагается исключительно на сериализацию объекта для сохранения результатов работы. Такой вид разработки в каком-то смысле и есть истинная цель архитектуры JavaBeans. Настоящее WYSIWYG-программирование. Однако исключительно визуальная разработка без возможности внедрения кода вручную, имеющейся в NetBeans, еще не доказала свою способность выйти за пределы простых приложений, применение исключительно визуальных сред программирования до сих пор не удалось продвинуть за рамки простых GUI экранных разметок.

Создание сериализованой версии кода

Если во время работы вы обратили внимание исходный код в NetBeans, вы могли заметить код, которые создавался при изменении свойств bean-компонентов. По умолчанию NetBeans создает методы, вызываемые для установки значений bean-компонентов в методе `initComponents()`. Например, если вы установили значение `42` для одного из полей `NumericField` bean-компонента в панели **Properties** (Свойства), оно будет встроено в приложение как начальное значение для bean-компонента вызовом метода `setValue()` во время инициализации. Но, если вы перейдете на вкладку **Code** (Код) на панели **Properties** (Свойства), то увидите, что доступны другие функции. Эта область содержит свойства, которые управляют тем, как NetBeans создает код приложения. Изменяя значение свойства **Code Generation** (Создание кода) с **Generate Code** (Создать код) на **Serialize** (Сериализовать), вы изменяете поведение NetBeans. Вместо создания вызова методов в исходном коде он сохранит ваш полностью сконфигурированный bean-компонент как сериализуемый и далее создаст соответствующий код для загрузки замороженного bean-компонента в приложение из файла.

Попробуйте изменить значение свойства **Code Generation** (Создание кода) на **Serialize** (Сериализовать) для bean-компонента `Juggler`. Перейдите в окно **Source** (Источник) и обратите внимание на метод `initComponents()`. Вы увидите строку, в которой для bean-компонента используется метод `static Beans.instantiate()` для загрузки его сериализованной копии.

NetBeans использует файл сериализованного bean-компонента как часть исходного кода и будет пересоздавать его при каждом сохранении исходного файла. Для запуска нашего примера мы должны в пер-

вую очередь выполнить руководство по сборке. Выбрав команду меню **Build ⇒ Build Main Project** (Выполнить ⇒ Запустить проект), вы сможете запустить файл, как раньше, используя пункт **Run File** (Запустить файл) контекстного меню. Причина явного создания заключена в стремлении NetBeans копировать сериализованный файл bean-компонента из директории исходных файлов в директорию скомпилированных классов (он должен быть достаточно разумным, чтобы выполнять это самостоятельно). Вы должны увидеть файл сериализованного bean-компонента с именем вида *LearnJava1_juggler1.ser* вместе с вашим исходным файлом (и развернутой директорией *classes*). Вы можете запустить пример и убедиться, что он действует в точности как версия сгенерированного кода.

Позже в этой главе мы подробнее обсудим работу с сериализованными bean-компонентами и попросим вас обратиться к этому сохраненному файлу bean-компонента.

Использование класса `BeanInfo`

Все это время все то, что NetBeans «знал» о наших bean-компонентах, было описано на низком уровне рефлексии, то есть через поиск методов наших классов. Класс `java.Beans.Introspector` собирает информацию о bean-компонентах использующих рефлексию в целях последующего анализа и описания, bean-компонента для любого инструмента, который хочет получить информацию о нем. Процесс самоанализа работает только при условии, что названия методов класса придерживаются соглашения имен JavaBeans, кроме того, он предоставляет некоторый контроль над тем, какие свойства и события точно отобразятся в меню NetBeans. Например, мы видели, что по умолчанию NetBeans показывает все, что мы наследовали от основного Swing-компонента. Это можно изменить, создав класс `BeanInfo` для наших bean-компонентов. Класс `BeanInfo` предоставляет инспектор JavaBeans, который выявляет информацию о свойствах, методах и событиях bean-компонента; мы даже можем использовать его для настройки представления текста, который появляется в меню NetBeans (и других IDE).

Класс `BeanInfo` реализует интерфейс `BeanInfo`. В большинстве случаев стандартное поведение `Introspector` оправдано. Вместо реализации интерфейса `BeanInfo`, мы расширяем класс `SimpleBeanInfo`, в котором реализованы все методы класса `BeanInfo`. Мы можем переопределить некоторые методы так, чтобы они предоставляли необходимую информацию, если вы не переопределяете метод, то `Introspector` действует

обычным образом. В нескольких следующих разделах мы разработаем класс `DialBeanInfo`, который будет предоставлять явную информацию о нашем bean-компоненте `Dial`.

Получение информации о свойствах

Начнем наше описание свойств `Dial`. Для этого мы должны реализовать метод `getPropertyDescriptors()`. Этот метод возвращает массив объектов `PropertyDescriptor`, по одному элементу для каждого свойства, которое мы хотим разгласить.

Для создания `PropertyDescriptor` вызываем его конструктор с двумя аргументами: имя свойства и класс. В следующем коде мы создаем дескрипторы для таких свойств `Dial`, как `value`, `minimum` и `maximum`. Далее мы вызываем методы класса `PropertyDescriptor` для предоставления дополнительной информации о каждом свойстве. Если наши методы были ограничены (сгенерирован `PropertyChangeEvents`, когда вносились изменения), мы должны вызвать метод `setBound()` из их класса `PropertyDescriptors`. Наш код подготовлен для получения `IntrospectionException`, которое может произойти, если что-то пойдет не так в ходе создания дескрипторов свойства, например при столкновении с несуществующим методом:

```
//файл: DialBeanInfo.java
package magicbeans;
import java.beans.*;
public class DialBeanInfo extends SimpleBeanInfo {
public PropertyDescriptor[] getPropertyDescriptors() {
try {
PropertyDescriptor value =
new PropertyDescriptor("value", Dial.class);
PropertyDescriptor minimum =
new PropertyDescriptor("minimum", Dial.class);
PropertyDescriptor maximum =
new PropertyDescriptor("maximum", Dial.class);
return new PropertyDescriptor [] { value, minimum, maximum };
}
catch (IntrospectionException e) {
return null;
}
}
}
```

Возможно, наиболее полезная вещь в `DialBeanInfo` — это то, что, предоставляя полную информацию о наших свойствах, мы автоматически скрываем остальные свойства, которые могли быть найдены при самоанализе. После компиляции `DialBeanInfo` и присоединения его к `Dial` вы увидите, что свойства его `JComponent` больше не отображаются в редакторе свойств NetBeans. (Если вы начинали работу с предкомпилированного JAR-файла из примера, то они могли не отображаться все это время.)

Класс `PropertyDescriptor` может предоставлять много другой информации о свойстве: имена методов доступа (если вы решили не использовать стандарт о соглашении имен), информацию о том, было ли свойство составлено, и класс, используемый как редактор свойств (если стандартные редакторы свойств не подходят).

Получение информации о событиях

Возможно, наибольшая польза от `DialBeanInfo` в том, что, предоставляя явную информацию о наших свойствах, мы автоматически скрываем другие свойства, которые могли быть найдены при самоанализе. После компиляции `DialBeanInfo` и создания пакета `Dial` вы увидите, что его свойство `JComponent` больше не отображается в редакторе свойств NetBeans. (Так было с самого начала, если вы начали со скомпилированного примера JAR.)

```
public EventSetDescriptor[] getEventSetDescriptors() {
try {
EventSetDescriptor dial = new EventSetDescriptor(
Dial.class, "dialAdjusted",
DialListener.class, "dialAdjusted");
dial.setDisplayName("Dial Adjusted");
return new EventSetDescriptor [] { dial };
}
catch (IntrospectionException e) {
return null;
}
}
```

В этом методе мы создаем объект `dial` класса `EventSetDescriptor`. Конструктор класса `EventSetDescriptor` получает четыре аргумента: класс, создающий событие, имя события (имя, которое отображается по

умолчанию в инструменте разработки), класс-слушатель и имя метода, к которому событие может быть направлено. Другие конструкторы позволяют вам связывать объет с интерфейсами слушателей, которые содержат несколько методов. После создания дескриптора мы вызываем метод `setDisplayName()` для установки понятного имени, которое будет отображаться в инструменте разработки таком, как NetBeans. То есть мы переопределяем имя, по умолчанию указанное в конструкторе.

Так как мы снабжаем дескрипторы свойств скрытыми свойствами, которые были обнаружены рефлексией, метод `EventSetDescriptors` может скрыть прочие свойства, которые были наследованы от класса основного компонента. Теоретически мы перекомпилировали `DialBeanInfo`, поместили его в JAR-файл и загрузили в NetBeans. Вы должны видеть только два события, которые мы явно описали: наши собственные `DialEvent` и `PropertyChangeEvent` (отображаемое как «DialAdjusted» и «Bound property change»). К сожалению, текущая версия NetBeans игнорирует эту информацию. Имея `EventSetDescriptor`, мы можем предоставить другой вид информации о событиях. Например, мы можем указать, что данное событие *одноадресное* — это означает, что оно может иметь только одного слушателя.

Добавление значков

Некоторые bean-компоненты, полученные в комплекте с NetBeans, обозначены на панели **Palette** (Палитра) привлекательными значками. Чтобы задать значок для разрабатываемого нами объекта `BeanInfo`, мы должны реализовать метод `getIcon()`. Вы можете присоединять до четырех значков размера 16×16 или 32×32 пикселов, цветных или монохромных. Далее показан метод `getIcon()` для класса `DialBeanInfo`:

```
public class DialBeanInfo extends SimpleBeanInfo {
...
public java.awt.Image getIcon(int iconKind) {
if (iconKind == BeanInfo.ICON_COLOR_16x16) {
return loadImage("DialIconColor16.gif");
} else
if (iconKind == BeanInfo.ICON_COLOR_32x32) {
return loadImage("DialIconColor32.gif");
} else
if (iconKind == BeanInfo.ICON_MONO_16x16) {
return loadImage("DialIconMono16.gif");
```

```
} else
if (iconKind == BeanInfo.ICON_MONO_32x32) {
return loadImage("DialIconMono32.gif");
}
return null;
}
```

Этот метод вызывается с константой, указывающей, какой тип значка запрашивается; например `BeanInfo.ICON_COLOR_16x16` запрашивает цветное изображение размером 16×16 пикселов. Если соответствующий значок существует, метод загружает изображение и возвращает объект `Image`. Если значок недоступен, метод возвращает значение `null`. Для удобства вы можете собрать изображения в JAR-файл, подобный файлу bean-компонента и его класса `BeanInfo`.

Хотя мы не используем их здесь, вы можете также использовать объект `BeanInfo` для предоставления информации о других открытых методах вашего bean-компонента, например свойствах со значениями-массивами и других функциях.

Создание настройщиков и редакторов свойств

JavaBeans позволяет обеспечить настройщик для ваших bean-компонентов. Настройщики являются объектами, которые выполняют расширенную настройку bean-компонентов в целом и позволяют предоставить собственный графический интерфейс для настройки bean-компонентов. Мы не можем рассказать, как написать настройщик, поскольку это выходит за рамки нашей темы, но это не так уж сложно. Достаточно сказать, что настройщик должен реализовывать интерфейс `java.beans.Customizer` и расширять свойство `Component` (или `JComponent`) так, чтобы он мог быть отображен.

Редакторы свойств — это способ предоставления дополнительных свойств. Например, вы могли реализовать редактор, чтобы получить возможность редактирования свойств, предназначенных для вашего bean-компонента. Вы могли предоставить редактор свойств, который позволит вам редактировать цену объекта в долларах и центах. Мы уже видели пару редакторов свойств, например, редактор со значениями свойства `Color`, в принципе такой же редактор вы можете создать самостоятельно. Дополнительно bean-компонент `Molecule` использует редактор для определения свойства `moleculeName`. *Редактор свойств* — не со-

всем настройщик, но его полное описание мы также не можем привести в рамках этой главы.То же касается и создания редактора свойств. Однако вам может помочь знание того, что редактор свойств должен реализовать интерфейс `PropertyEditor`; обычно это делается расширением класса `PropertyEditorSupport`, который, по умолчанию, реализует большинство методов.

Программирование вручную с применением bean-компонентов

Мы долго рассматривали создание и использование bean-компонентов средствами разработки bean-приложений. Это основная мотивация JavaBeans, по крайней мере в разработке GUI. Но bean-компоненты не ограничены использованием автоматизированных инструментов. Нет причин, по которым вы не могли бы вручную программировать bean-компоненты.

Вы могли бы использовать разработчик для создания bean-компонентов пользовательского интерфейса вашего приложения и последующей загрузки, сериализующей bean-компоненты или их коллекцию в вашем коде. Сейчас мы покажем такой пример.

Установка bean-компонентов и управление типами

Bean-компоненты — это абстракция простых Java-классов. Они добавляют, по соглашению, функции, не являющиеся частью языка Java. Для включения определенных дополнительных возможностей JavaBeans, мы используем специальные инструменты, которые замещают основные операции языка. Конкретно, при работе с bean-компонентами, нам обеспечена замена трех основных Java-операций: создание объекта с использованием оператора `new`, проверка типа объекта с использованием оператора `instanceof` и отбор по типу с применением выражения `cast`. Вместо этих операций мы используем соответствующие статические (`static`) методы класса `java.beans.Beans`, представленные в табл. 22.1.

Табл. 22.1. Методы класса `java.beans.Beans`

Оператор	Эквивалент
`New`	`Beans.instantiate(classloader, name)`
`Instanceof`	`Beans.isInstanceOf(object, class)`

`Beans.instantiate()` — это операция `new` для bean-компонентов. Он получает загрузчик класса и имя класса bean-компонента или сериализованый bean-компонент как аргументы. Это дополнение простого оператора `new`, которое также может загружать bean-компоненты в сериализованной форме. Если вы используете метод `instantiate()`, вы не должны указывать дополнительно, как вы будете использовать bean-компонент — как класс или как сериализованный объект. Метод `instantiate()` сначала пытается загрузить исходный файл ориентируясь на имя bean-компонента, превращая имена пакетного-стиля (с точками) в имена пути (со слешами) и расширением *.ser*. Например, `magicbeans.NumericField` преобразуется в *magicbeans/NumericField.ser*. Если сериализованная форма bean-компонента не найдена, метод `instantiate()` пытается создать экземпляр класса по имени*.

Методы `Beans.isInstanceOf()` и `Beans.getInstanceOf()` выполняют задачи проверки типов bean-компонентов и преобразования их в новый тип. Эти методы были предназначены, чтобы осуществить совместную работу одного или более bean-компонентов при реализации «виртуальных» или динамических типов. Они должны позволять bean-компонентам управлять этим поведением, предоставляя различные собственные «представления». Однако они пока не добавляют функциональности и не применяются широко.

Работа с сериализованными bean-компонентами

Помните bean-компонент `Juggler`, который мы сериализовали не так давно? Мы предполагаем, что вы сохранили `Juggler`, выбрав пункт **Serialization** (Сериализовать), когда работали с классом LearnJava1, и что NetBeans, следовательно, сохранил его в файл *LearnJava1_juggler1.ser*. Если вы не сделали этого, вы можете использовать следующий фрагмент кода для сериализации bean-компонента в файл на ваш выбор:

```
// Сериализация экземпляра Juggler в файл...
import magicbeans.sunw.demo.juggler.Juggler;
import java.io.*;
public class SerializeJuggler {
public static void main( String [] args ) throws Exception
{
```

* Эта функция, казалось бы, применима к XML-сериализации bean-компонентов, с использованием и XMLOutputStream, но это в настоящее время это не реализовано. Это другой знак стагнации, присущей JavaBeans API.

```
Juggler duke = new Juggler();
ObjectOutputStream oout = new ObjectOutputStream(
new FileOutputStream("juggler.ser") );
oout.writeObject( duke );
oout.close();
}
}
```

Имея замороженный компонент `Juggler`, скомпилируйте следующее небольшое приложение:

```
//файл: BackFromTheDead.java
import java.awt.Component;
import javax.swing.*;
import java.beans.*;
public class BackFromTheDead extends JFrame {
public BackFromTheDead( String name ) {
super("Возрожденный Bean-компонент!");
try {
Object bean = Beans.instantiate(
getClass().getClassLoader(), name );
if ( Beans.isInstanceOf( bean, JComponent.class) ) {
JComponent comp = (JComponent)
Beans.getInstanceOf(bean, JComponent.class);
getContentPane().add("Center", comp);
} else {
System.out.println("Bean-компонент не является JComponent...");
}
} catch ( java.io.IOException e1 ) {
System.out.println("Ошибка загрузки сериализованного объекта");
} catch ( ClassNotFoundException e2 ) {
System.out.println(
"Невозможно найти класс, связанный с объектом ");
}
}
public static void main(String [] args) {
JFrame frame = new BackFromTheDead( args[0] );
frame.pack();
frame.setVisible(true);
}
}
```

Запустите эту программу, передав имя вашего сериализованного объекта в качестве аргумента, и убедитесь, что *magicbeans.jar* находится в вашей директории classpath. Имя не должно включать расширение *.ser*; метод `Beans.instantiate()` добавит его автоматически при поиске сериализованного объекта или версии класса. Жонглер должен вернуться к жизни, жонглируя вновь, как показано на рис. 22.9.



Рис. 22.9. Сериализованный Juggler

В `BackFromTheDead` мы используем метод `Beans.instantiate()` для загрузки нашего сериализуемого bean-компонента по имени. Далее мы проверяем, является ли он GUI-компонентом, используя метод `Beans.isInstanceOf()`. (Это так, потому что `Juggler` —подкласс `java.awt.Component`.) Наконец, мы добавляем экземпляр объекта к `Component`, используя метод `Beans.getInstanceOf()`, и добавляем его к `JFrame` нашего приложения. Отметим, что нам до сих пор необходимо выражение `cast`, чтобы настроить класс `Object` на возвращение методом `getInstanceOf()` в `JComponent`. Это выражение может казаться сложным, но это мост между динамическим и статическим поиском bean-компонента, проверяющий типы во время компиляции.

Все, что мы сделали сейчас, могло быть выполнено с использованием простого класса, `java.io.ObjectInputStream`, описанного в главе 12. Но такие методы разработки bean-компонентов предназначены для того, чтобы пользователю не приходилось вникать в то, как они реализуются и хранятся.

Еще кое-что, перед тем как мы продолжим. Мы отметили, что когда `Juggler` был восстановлен, bean-компонент продолжил жонглировать. Это подразумевает, что потоки были запущены, когда bean-компонент был десериализован. Сериализация не управляет автоматически переходящими ресурсами, такими как потоки или даже загрузка изображений, но нетрудно взять контроль в свои руки, чтобы завершить процесс реконструкции состояния bean-компонента, когда он будет десериализован. Взгляните на исходный код программы `Juggler` (доступный вместе с этим примером) и обратитесь к главе 12, где вы найдете обсуждение десериализации объектов методом `readObject()`.

Отслеживание событий выполнения с применением рефлексии

Мы обсудили, как инструменты разработки используют рефлексию для анализа классов. Сегодня рефлексия часто применяется таким способом в приложениях для выполнения динамической активности, которая была бы невозможна без нее. В этой части мы рассмотрим динамическое отслеживание событий, которые могут быть сконфигурированы при выполнении.

В главе 16 мы видели, как классы адаптеров могут быть созданы для соединения действий событий со сложными методами нашего кода, что позволяет нам четко разделять GUI и логику приложения. В этой главе мы увидим, как NetBeans добавляет этот код адаптера между bean-компонентами для автоматического выполнения данной задачи.

Модель событий AWT/Swing уменьшает необходимость в создании подкласса компонентов для выполнения простых отслеживаний. Если мы начнем полагаться на специальные классы адаптеров, мы можем быстро достичь того же количества адаптеров, что и компонентов. Анонимные внутренние классы позволяют нам скрывать эти классы, но они все еще остаются. Потенциальное решение для больших или специализированных приложений: создать общий адаптер событий, который обслуживает набор источников событий и целей одновременно.

Класс `java.beans.EventHandler` — это динамический диспетчер событий, который просто вызывает методы для соответствующих событий. Необычным `EventHandler` делает то, что это первая стандартная утилита использующая рефлексию, что позволяет определять метод по *имени*. Другими словами, вы просите `EventHandler` направить событие в обработчик, указывая объект-обработчик и строку, содержащую имя метода для вызова этого объекта.

Мы можем использовать метод `create()` класса `EventHandler`, чтобы получить адаптер для указанного типа прослушивателя события, определяя целевой объект и имя метода, для вызова при наступлении события. Целевой объект не должен быть прослушивателем для конкретного типа события или других конкретных видов объектов. Следующее приложение, `DynamicHookup`, использует `EventHandler` для связи кнопки с методом `launchTheMissiles()` нашего класса:

```
//файл: DynamicHookup.java
import javax.swing.*;
import java.awt.event.*;
```

```
import java.beans.EventHandler;
public class DynamicHookup extends JFrame {
JLabel label = new JLabel( "Готов...", JLabel.CENTER );
int count;
public DynamicHookup() {
JButton launchButton = new JButton("Запуск!");
getContentPane().add( launchButton, "Юг" );
getContentPane().add( label, "Центр" );
launchButton.addActionListener(
(ActionListener)EventHandler.create(
ActionListener.class, this, "launchTheMissiles"));
}
public void launchTheMissiles() {
label.setText("Запущено: "+ count++ );
}
public static void main(String[] args) {
JFrame frame = new DynamicHookup();
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setSize(150, 150);
frame.setVisible( true );
}
}
```

Здесь мы вызываем метод `create()` класса `EventHandler`, передавая в `ActionListener` класс, ссылку на объект (`this`) и строку с именем метода для вызова по ссылке, при наступлении события. `EventHandler` создает прослушиватель для соответствующего типа и регистрирует информацию нашей ссылки. Так мы не только исключаем внутренний класс, реализация `EventHandler` позволяет ему внутренне разделять адаптеры, создавая небольшое количество объектов.

Этот пример демонстрирует, как мы могли бы вызвать метод, не принимающий аргументов, но `EventHandler` на самом деле может сделать больше, установив соответствие событий и свойств JavaBeans. Следующая форма метода `create()` инструктирует `EventHandler` вызвать метод `launchTheMissiles()`, передавая свойство `source` в качестве аргумента:

```
EventHandler.create(
ActionListener.class, target, "launchTheMissiles", "source")
```

Все события получают свойство `source` (через метод `getSource()`), но мы можем пойти дальше, определяя цепочку разделенных точками

«get» свойств, которые применяются прежде, чем значение передается методу. Например:

```
EventHandler.create(
ActionListener.class, target, "launchTheMissiles", "source.text")
```

Параметр `source.text` вызывает значение `getSource().getText()` для передачи в качестве аргумента в метод `launchTheMissiles()`. В нашем примере это могла бы быть надпись на кнопке. Другие формы метода `create()` предоставляют больше гибкости в выборе одного из множества методов в интерфейсе прослушивателя, а также других опций.

Как это работает

`EventHandler` использует `java.lang.reflect.Proxy` — фабрику, которая может создавать адаптеры, реализующие интерфейсы любого типа во время выполнения.

Определив один или несколько интерфейсов прослушивания событий, например `ActionListener`, мы получили адаптер, который реализует интерфейсы этих прослушивателей, создаваемых на лету.

Адаптер — это специально созданный класс, который передает все вызовы методов их интерфейса назначенному объекту `InvocationHandler`. См. подробности о промежуточных классах в главе 1.

BeanContext и BeanContextServices

Мы много говорили о сложных механизмах объединения компонентов Java bean во время проектирования и выполнения. Однако мы ничего не сказали о средах, в которых существуют компоненты Java bean. Для создания сложного расширяемого приложения мы выбирали способ обнаружения компонентов Java bean друг другом или «рандеву» во время выполнения.

Пакет `java.beans.beancontext` предоставляет данный тип контейнера окружающей среды. Он также предоставляет основной механизм поиска «сервисов» для bean-компонентов, которые хотят сообщить о своих возможностях. Эти механизмы использовались некоторое время, но не нашли широкого применения в стандартных пакетах Java. Все же они остаются интересным и важным средством, которое вы можете использовать в собственных приложениях.

Java Activation Framework

Java Activation Framework (JAF) может использоваться bean-компонентами, работающими со множеством внешних типов данных, таких как медиа, полученные из файлов и потоков. По существу, это обобщенный механизм JavaBeans для обработки контекста/протокола. Пакет `javax.activation` содержит расширяемый набор классов, которые оборачивают произвольные, неподготовленные источники данных, чтобы предоставить доступ к своим данным в виде потоков или объектов, определить MIME-тип данных и перечислить зарегистрированный набор «команд» обработки данных.

JAF предоставляет два основных интерфейса: `DataSource` и `DataHandler`. Интерфейс `DataSource` действует подобно обработчику протокола, который мы обсуждали в главе 14. Он оборачивает источник данных и определяет MIME-типы для потока данных. Интерфейс `DataHandler` действует подобно обработчику контекста, за исключением того, что он предоставляет на порядок больше возможностей, чем просто доступ к данным. Интерфейс `DataHandler` разработан для обертывания `DataSource` и представления данных в различных формах. Он также предоставляет список управляющих команд, которые могут быть использованы для доступа к данным. Кроме того `DataHandler` реализует интерфейс `java.awt.datatransfer.Transferable`, что позволяет передавать данные между компонентами приложения четко определенным способом.

Технология JavaBeans и фреймворки промышленного уровня, основанные на обычных объектах

Enterprise JavaBeans (EJB) — это очень обширная тема, но мы можем уделить ей лишь несколько строк. Если вам необходимо больше информации, см. книгу «Enterprise JavaBeans» Ричарда Монсона Хейфела (Издательство O'Reilly). По сути EJB перенимает философию переносимых, заменяемых компонентов и распространяет ее на серверную сторону, подстраивая ее под нужды многоуровневых, сетевых и работающих с базами данных приложений. Хотя EJB придерживается основ концепции JavaBeans, он на порядок крупнее и более специализирован. EJB имеет мало общего с тем, о чем мы говорили в этой главе. EJB — это серверные компоненты для сетевых приложений. И EJB-компоненты,

и простые компоненты Java bean — это многоразовые, переносимые компоненты, которые могут быть развернуты и настроены в конкретной среде. Но EJB-компоненты инкапсулируют доступ к бизнес-логике и таблицам данных, а не к GUI и элементам программы.

EJB связывает набор других системно-ориентированных Java API, включая доступ к базе данных, транзакции и службу именования, в единую компонентную модель серверного приложения. EJB предъявляет больше требований к структуре текста программы, чем более простые компоненты Java bean. Это сделано для того, чтобы позволить *EJB-контейнеру* серверной стороны вместить в себя множество возможностей и оптимизировать деятельность вашего приложения, не прибегая к написанию большого объема исходного кода программы.

Первые два основных релиза *Java Enterprise Edition*, включающие EJB, по мнению многих разработчиков, были чрезмерно сложными. Поддержка EJB требовала большого числа классов, интерфейсов и конфигурационных файлов. В результате многие популярные фреймворки с открытым исходным кодом, использующие Plain Old Java Objects (POJOs) для выполнения схожих задач, быстро заняли видное положение. POJO — это объекты Java, не требующие следования каким-либо соглашениям, реализации каких-либо интерфейсов или наследования от какого-либо основного класса, хотя на практике они часто оказываются объектами JavaBeans. Наиболее известный из POJO-фреймворков — Hibernate, фреймворк для отображения состояния объекта Java в базу данных, и Spring, фреймворк для разработки корпоративных приложений общего назначения.

Материал, защищенный авторским правом

Глава 23
АППЛЕТЫ

Вначале одним из перспективных направлений Java были приложения, способные поступать на ваш компьютер из сети при необходимости. На смену времени, когда люди приобретали упакованные в целлофан коробочные версии текстового редактора, а затем устанавливали его и обновляли каждые несколько лет, пришло время, когда стало возможным загружать программное обеспечение напрямую из Интернета, безопасно и на любую платформу. Сегодня мы принимаем распространение программного обеспечения по сети как должное. Мобильные устройства привели к появлению магазинов приложений, для установки программного обеспечения, и многие (если не большинство) из наиболее используемых приложений теперь существуют исключительно на браузерной основе. К огорчению фанатов Java, эти развитие этих достижений пошло другими путями, возможно, менее ясными и быстрыми, чем предлагаемые платформой Java.

Эта глава посвящена API (интерфейс программирования приложений, application programming interface, API) *апплетов*, которые были ранним механизмом Java для передачи приложений в веб-браузер. Сегодня апплеты не широко используются, но являются частью основы и до сих пор хорошо работают в некоторых нишах. Полезно понять базовую функциональность апплетов прежде, чем у вас появится желание их создавать. В процессе рассмотрения апплетов, попутно мы также затронем общие области, которые представляют интерес, такие как система безопасности Java.

Апплет — это программа, написанная на языке Java, выполняемая в контексте веб-страницы. Так же как за изображением или ссылкой, за апплетом закреплена некоторая прямоугольная область на экране пользователя. Веб-браузер знает, как загрузить классы для апплета и запустить их, когда загружает страницу, содержащую Java-апплет. В этой части описывается, как апплеты работают и как они взаимодействуют с веб-страницами. Мы также кратко расскажем о Java-плагинах и связанных с ними технологиях, таких как Java Web Start.

Политика браузерных приложений

Для начала немного истории. Потенциал апплетов по добавлению динамического содержимого в веб-страницы был одной из движущих сил распространения языка Java. До введения в 1994 году языка Java, не существовало стандартных способов реализации этого; даже повсеместно теперь использующиеся анимированные GIF-изображения еще не поддерживались широко. Первым поддерживающим апплеты браузером на основе Java стал HotJava разработанный Sun. Это было оригинальное «сенсационное» Java-приложение. Позже, в 1995 году, Netscape анонсировали поддержку Applet API в их браузерах, и уже позже к массовому движению присоединился Microsoft. Какое-то время казалось, что Java может стать инструментом будущего во Всемирной паутине, но этот путь не был столь гладким.

Множество проблем, как технических, так и политических, сопровождало первые годы использования Java в браузерах и клиентских приложениях. Для столь молодой платформы проблемы производительности были предсказуемы. Но что действительно калечило ранние версии Java, так это непереносимый и содержащий ошибки AWT — оригинальный инструментарий графического пользовательского интерфейса. Многие люди игнорируют тот факт, что успех Java в качестве портативного языка в большей степени результат того, что многие Java API реализованы на Java. Вы, возможно, будете удивлены, узнав, как много native-приложений включают в себя код Java: все от криптографии до DNS (*Domain Name System* — система доменных имен) выполнено на Java и не требует переноса на новые платформы. Точно так же, возрождение приложений Java GUI, которое прослеживалось в последующие годы, было полностью связано с введением чистого Java-инструментария GUI Swing. Оригинальная система AWT была основана на родном коде, который подключался к каждой новой системе, с учетом тонкой и сложной зависимости от платформы. Библиотека AWT фактически была большой, графической C-программой, которую Java передавал через набор интерфейсов, и в какой-то степени язык Java был несправедливо освещен, как непереносимый и обладающий ошибками ассоциации.

Язык Java столкнулся и с другими, менее техническими препятствиями. К примеру, Netscape навязал оригинальную библиотеку AWT всему миру, в то время, как Java был реализован, как «истинно полиэкранный» вместо менее мощного, но действительно портатив-

ного, графического инструментария. Позже они представили чистый инструментарий Java GUI, названный IFC (предшественник Swing), который боролся, чтобы набрать обороты. Позже Microsoft прочно закрепил нас на AWT, заморозив уровень Applet API в своих браузерах на версии Java 1.1. Апплеты томились с исходным GUI, пока между Sun и Microsoft шли судебные разбирательства на протяжении 1990-х. В результате поддержка апплетов в веб-браузерах оставалась в хаотичном состоянии.

Корпорация Sun сделала попытку изолировать Java от битвы браузеров, введя Java-плагины. Плагины позволяют апплетам запускаться на обновленной Java VM, но на тот момент не хватало критической массы и почти повсеместного распространения, которое требовалось, чтобы сделать плагины полезными. Более слабый, но более распространенный плагин Adobe Flash процветал, так как разработчики искали способ разворачивать серьезные приложения в браузере. В 2000-х JavaScript и HTML (*HyperText Markup Language*, язык гипертекстовой разметки) достаточно развились и сделали сами браузеры более жизнеспособной платформой для разработки программного обеспечения, что привело к большому заблуждению, будто JavaScript медленнее и обладает меньшими возможностями по сравнению со своим тезкой Java, де-факто ставшим языком веб-программирования.

Немного расстроены? Не принимайте близко к сердцу, JavaScript был включен в серьезные платформы клиентской стороны, Flash постепенно исчезает, а Java доминирует в разработке серверной стороны и делает возможными портативные библиотеки, которые во многом пробудили интернет-революцию. Трудно сказать, что ждет нас в будущем, но Java наверняка останется в игре еще очень надолго.

Поддержка апплетов и Java-плагины

Как мы упоминали ранее, состояние поддержки Java в веб-браузерах всегда пребывало в беспорядке. Наиболее надежным способом запустить Java в веб-браузере всегда оставался специальный плагин. Плагин — это бесплатный, устанавливаемый пользователем компонент (различно реализованный для каждого браузера), который поддерживает Java. Использование плагина реализует в конечной перспективе полноценную поддержку в браузере, используя различные пакеты программного обеспечения, которые могут быть установлены и обновлены независимо от версий браузера.

Недавно некоторые платформы, особенно OS X, полностью отвергли встроенную поддержку Java в браузере, предоставляя пользователям возможность самостоятельно, при их желании, устанавливать Java-плагин для запуска апплетов. Это не идеал, но упрощает процесс, исключая некоторую двусмысленность. Если у вас установлена Java 7 на OS X или Windows, то вы у вас должен быть также и Java-плагин для запуска апплетов, обсуждаемых в этой главе. В некоторых случаях первое время при попытке просмотреть страницу, содержащую апплет, вам будет предложено включить Java в вашем браузере. Вы также можете обратиться к «панели управления» Java в Windows или к «панели настроек» в OS X для дополнительной настройки Java после установки Java 7.

Класс `JApplet`

Класс `JApplet` — это Swing-компонент `JPanel` целевого назначения. Это GUI-контейнер, имеющий некоторую дополнительную структуру, которая позволяет использовать ее в веб-браузерах «сторонних» разработок. Апплеты также имеют собственный жизненный цикл, который позволяет им действовать скорее подобно приложению, нежели статическому компоненту. Хотя апплеты обычно бывают относительно простыми, их сложность ничем не ограничена, кроме как проблемами загрузки и кеширования их контента. Исторически апплеты стремились быть небольшими «виджетами».

Класс `javax.swing.JApplet` определяет функциональность ядра апплета (`java.awt.Applet` старше, чем основные на AWT формы.)

По своей структуре апплет — это оболочка для вашего Java-кода. В отличие от отдельных графических Java-приложений, которые начинают выполняться с метода main() и создают GUI, апплет — это компонент, ожидающий, что он будет включен в некоторый дополнительный GUI. Таким образом, апплет не может запуститься самостоятельно, он запускается в контексте веб-браузера или специальной программы `appletviewer`, о которой мы расскажем позже. Вместо того, чтобы ваше приложение создавало `Jframe` для хранения вашего GUI, вы создаете ваше приложение внутри объекта `JApplet` (который является наследником класса `Container`) и позволяете браузеру добавить ваш апплет на страницу.

Апплет размещается на веб-странице в HTML-теге `<applet>`, который мы рассмотрим позже в этой главе. В самом простом случае вам

необходимо лишь указать имя класса апплета и размер апплета в пикселах:

```
<applet code="AnalogClock" width="100" height="100"></applet>
```

С прагматической точки зрения апплет внедряется в другую разработку, и поэтому к нему относятся с подозрением. Веб-браузеры, которые запускают апплеты, накладывают ограничения на их действия. Ограничения вводятся управлением безопасности апплетов. Браузеры предоставляют все, что необходимо апплету, через их контекст — API, используемый апплетом для взаимодействия с окружающей средой.

Компонент `JApplet` настроен для встраивания в страницу и использования в среде просмотра, которая объединяет его с ресурсами. Однако во всех других отношениях апплет — это лишь обычный объект `Panel`. На рис. 23.1 показан апплет в виде `Panel`. Компонент `JApplet` может содержать компоненты пользовательского интерфейса и использовать все базовые возможности рисования и обработки событий класса `Component` как любой другой объект `Panel`. Вы можете рисовать на объекте `JApplet`, переопределяя метод `paint()`, и реагировать на события в области отображения `JApplet`, предоставляя соответствующие обработчики событий. Апплеты также имеют дополнительную структуру, которая помогает им взаимодействовать с браузерным окружением.

Никаких других отличий, кроме структуры верхнего уровня и ограничений безопасности, между апплетами и приложениями нет. Если ваше приложение жизнеспособно при ограничениях накладываемых системой безопасности браузера, вы можете преобразовать его так, чтобы оно действовало и как апплет, и как отдельное приложение. Обычно вы используете сам класс апплета только как оболочку для управления жизненным циклом и представлением вашего приложения: создание GUI, запуск и остановка. Другими словами, основная часть вашего кода должна легко адаптироваться к разработке и апплета, и отдельного приложения.

Жизненный цикл апплета

Класс `Applet` содержит четыре метода, которые могут быть переопределены, для введения их в его жизненный цикл. Методы `init()`, `start()`, `stop()`, и `destroy()` вызываются программой просмотра апплетов или веб-браузером, чтобы перехватить поведение апплета. Метод `init()` вызывается единственный раз, после создания апплета. В методе `init()` вы

представляете основные установки, такие как параметры структурного анализа, создания пользовательского интерфейса и загрузки ресурсов.

Условно, апплеты не предоставляют никаких конструкторов для выполнения любых настроек, поскольку конструктор подразумевает, что он будет вызван средой апплета, для простоты создания. Это может произойти до того, как апплет получит доступ к требуемым ресурсам, например информации о его окружении. Поэтому апплеты обычно не выполняют здесь никаких действий; вместо этого они должны зависеть от конструктора класса `JApplet` и выполняют его инициализацию в методе `init()`.

Метод `start()` вызывается везде, где виден апплет; не должно вызвать удивления и то, что метод `stop()` вызывается везде, где апплет становится невидимым.

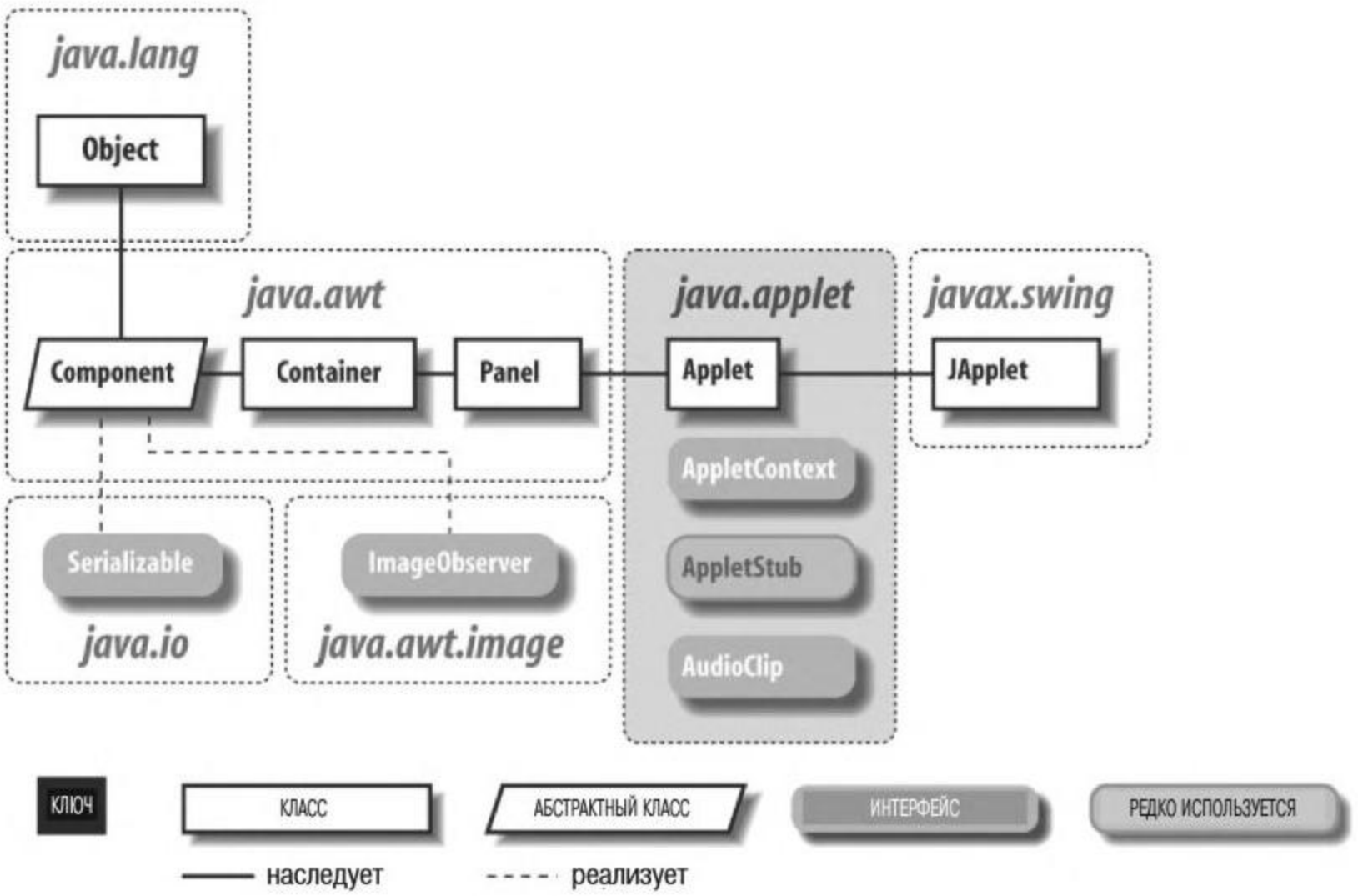

Рис. 23.1. Пакет `java.applet`

Метод `init()` вызывается один раз в жизненном цикле апплета, в то время как методы `start()` и `stop()` вызываются произвольное количество раз (хотя всегда в логической последовательности). Метод `start()` вызывается при отображении апплета как, например, при прокрутке экрана, а метод stop(), когда прокрутка завершена или если посетитель покидает документ. Метод `start()` сообщает апплету, что он должен быть активен. Апплет может захотеть создать потоки, анимацию или

умничать, выполняя полезную (или раздражающую) работу. Вызывая метод `stop()` мы даем апплету понять, что нужно бездействовать. Апплет должен прекратить нагружать центральный процессор или вести расточительную активность, когда он остановлен, и возобновить работу, когда (или если) будет вновь запущен. Однако нет требования, чтобы невидимый апплет прекратил вычисления; в некоторых приложениях может быть полезным, чтобы апплет продолжил вычисления в фоновом режиме. Просто будьте внимательны к пользователям, которые не хотят, чтобы скрытый апплет снижал возможности системы.

В завершение, метод `destroy()` дает апплету последний шанс очиститься перед его удалением, спустя некоторое время после последнего вызова метода `stop()`. Например, апплет мог пожелать любезно закрыть используемые каналы связи. Момент вызова метода `destroy()` зависит от браузера; Netscape вызывает метод `destroy()` только перед последующим удалением апплета из кеша. Это значит, что, хотя апплет может цепляться за жизнь после вызова метода `stop()`, как долго это может продолжаться — непредсказуемо. Если вы хотите поддерживать апплет, пока пользователь посещает другие страницы, вам необходимо поместить его в HTML-фрейм, тогда, если апплет станет невидимым, не произойдет вызова метода `stop()` (см. раздел «Устойчивость и навигация апплетов» далее).

Если вы читали эту книгу с самого начала, то уже пару раз встречались с апплетами в других темах. В главе 9 мы создали апплет простых часов, в главе 13 мы использовали апплет, чтобы передать пакеты информации через веб-браузер. Сейчас испытаем пример использования `JApplet` на основе Swing. Следующий пример, `ShowApplet`, изображенный на рис. 23.2, не делает ничего особенного, но вы можете использовать его для проверки версии Java, которая запущена в вашем браузере (и увидеть какие плагины установлены), и посмотреть, когда апплет запускается и завершается.

```
import javax.swing.*;
import java.awt.event.*;
public class ShowApplet extends JApplet {
JTextArea text = new JTextArea();
int startCount;

   public void init() {
```

```
JButton button = new JButton("Нажми меня");
button.addActionListener( new ActionListener() {
public void actionPerformed( ActionEvent e ) {
text.append("Button Pressed!\n");
}
} );
getContentPane().add( "Центр", new JScrollPane( text ) );
JPanel panel = new JPanel();
panel.add( button );
getContentPane().add( "Юг", panel );
text.append( "Версия Java: "
   +System.getProperty("java.version")+"\n" );
   text.append( "Апплет вызвал метод init()\n" );
}
public void start() {
text.append( "Апплет запущен: "+ startCount++ +"\n" );
}
public void stop() {
text.append( "Апплет остановлен.\n" );
}
}
```

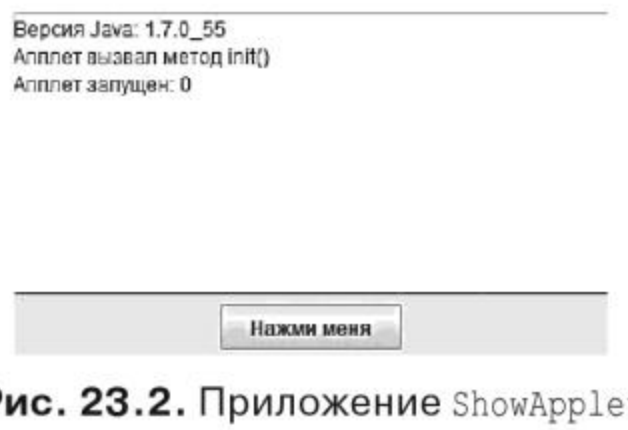

Рис. 23.2. Приложение `ShowApplet`

После компиляции апплета, мы должны создать HTML-страницу, в которой он будет размещен. Мы сделаем следующее:

```
<html><head><title>ShowApplet</title></head>
<body>
<applet code="ShowApplet" width="300" height="300">
Ваш браузер не поддерживает Java.</applet>
</body>
</html>
```

Позже в этой главе мы подробнее обсудим теги апплета и другие вопросы, связанные со встраиванием апплета в документы. Сейчас просто сохраним этот код в файле *showapplet.html*. Загрузите созданный файл, используя ваш любимый веб-браузер, и посмотрите, что произойдет. (Мы предполагаем, что к этому моменту на вашем компьютере установлена платформа Java; возможно, вам необходимо включить Java в вашем браузере, чтобы увидеть апплет, если она отключена по умолчанию.) Если у вас есть доступ к веб-серверу, вы можете использовать его. С другой стороны, вы можете открыть файл локально, используя или команду меню браузера **`Open File`** (Открыть файл) или URL-адрес, как здесь:

```
file:///Users/somedir/showapplet.html
```

Апплет показывает версию Java, в которой он запущен, и выводит сообщение, когда кнопка нажата. Если у вас установлена последняя версия плагина Java, вы должны увидеть в окне «Версия Java: 1.7», вне зависимости от того, какой браузер вы используете (включая Microsoft Internet Explorer).

Апплет выводит сообщения при вызове методов `start()` и `stop()` с подсчетом количества вызовов. Вы можете поэкспериментировать с компоновкой страницы, используя браузеры различной конфигурации, чтобы увидеть, когда ваш апплет перегружается или перезапускается. Если браузеру не удается отобразить апплет с верной версией Java, не беспокойтесь. Позже в этой главе мы поговорим о том, как преобразовать HTML для ускорения браузера при явном использовании плагина Java.

«Песочница» безопасности для апплетов

Апплеты помещаются в карантин в браузере апплетом `SecurityManager`. `SecurityManager` — это часть веб-браузера или программы просмотра апплетов. Он уже установлен до того, как браузер загрузит какой-либо апплет, и реализует базовые ограничения, которые позволяют пользователю безопасно запускать не доверенные апплеты (загруженные через Интернет). Помните, что в отдельных приложениях Java нет никаких неотъемлемых ограничений безопасности. Действия апплетов ограничивает браузер, используя политику безопасности.

Большинство браузеров накладывают следующие ограничения на недоверенные апплеты:

- недоверенные апплеты не могут читать и записывать файлы на локальном хосте.
- недоверенные апплеты могут открывать сетевые соединения (сокеты) только с сервером, с которого они были загружены.
- недоверенные апплеты не могут запускать другие процессы на локальном хосте.
- недоверенные апплеты не могут иметь собственных методов.

Причина таких ограничений очевидна: наверняка вы не хотите, чтобы программа, загруженная со случайного интернет-сайта, получила доступ к вашим файлам или запускала произвольные программы. Хотя непроверенные апплеты не могут получить прямой доступ к чтению и записи файлов на клиентской стороне или общаться с произвольным узлом в сети, они могут работать с сервером, чтобы хранить данные и взаимодействовать. Например, апплет может использовать возможность Java RMI для выполнения процессов на своем сервере. Апплет может взаимодействовать с другими апплетами в сети, используя этот сервер в качестве прокси-сервера.

Доверенные апплеты

Мы использовали термин «недоверенные апплеты», так что для вас не должно быть сюрпризом существование такого понятия, как доверенные апплеты. Апплеты становятся доверенными, при использовании цифровой подписи для *jar*-файла, содержащего их код. Так как подпись идентифицирует владельца апплета однозначно, пользователь может различать доверенные апплеты (например, апплеты, которые вы получили с сайта или от лица, которому вы доверяете, не могут сделать ничего разрушительного) и не доверенные апплеты. В среде браузера, который поддерживает цифровые подписи, доверенные апплеты могут получить больше разрешений на выход за пределы «песочницы» безопасности апплетов. Доверенные апплеты могут выполнять все те же действия, что и отдельные приложения Java: читать и записывать файлы, устанавливать соединения с произвольной машиной и взаимодействовать с операционной системой для запуска процессов. Доверенные апплеты все равно не могут иметь собственных методов, а включение таких методов в апплет все равно уничтожило бы его портативность.

Так как подписанные апплеты теперь являются отдельной увлекательной темой, мы не будем больше говорить о них в этой главе. Если

вам необходимо больше информации о них, пожалуйста, загрузите файл eksmo.ru/smv/Primers.zip, в котором находятся исходные коды примеров к книге.

Обзор ресурсов апплетов

Апплет должен взаимодействовать со своим браузером или программой просмотра апплетов. Например, ему может потребоваться конфигурация параметров HTML-документа, в котором он появляется. Возможно, апплету также необходимо загрузить изображения, аудио-записи или другие ресурсы. Кроме того, возможно, апплет хочет спросить программу просмотра апплетов о других апплетах, размещенных на этой же HTML-странице, для взаимодействия с ними. Чтобы получить ресурсы из окружающей среды разработки, апплеты используют интерфейсы `AppletStub` и `AppletContext`, предоставляемые браузером.

Параметры апплетов

Апплет может получить конфигурацию параметров из тегов `<param>`, размещенных внутри тега `<applet>` в HTML-документе, мы обсудим их позже. Вы можете получить эти параметры, используя метод `getParameter()` класса `Applet`. Например, следующий код считывает параметры `imageName` и `sheep` HTML-страницы:

```
String imageName = getParameter( "imageName" );
try {
int numberOfSheep = Integer.parseInt( getParameter( "sheep" ) );
} catch ( NumberFormatException e ) { /* использовать предопределенный */
}
```

Доступ к параметрам предоставляет API, который позволяет апплету получать информацию (справку) о них. Метод `getParameterInfo()` может возвращать массив массивов строк, списков и описаний параметров апплета.

Ресурсы апплета

Апплет может узнать свое расположение, используя методы `getDocumentBase()` и `getCodeBase()`. Метод `getDocumentBase()` возвращает основной URL-адрес документа, в котором появляется апплет, метод `getCodeBase()` возвращает основные URL-адреса файлов класса `Applet`

(есть два похожих файла). Апплет может использовать эти методы для составления относительных URL, с которых загружаются другие ресурсы с их сервера, такие как изображения, звук и другие данные. Метод `getImage()` получает URL и запрашивает изображение из среды просмотра. Изображение может быть кешировано или загружено позже при использовании. Метод `getAudioClip()` аналогичен, но работает с файлами звука.

В следующем примере метод `getCodeBase()` используется для построения URL и загрузки свойств конфигурационного файла, размещенного в той же удаленной директории на веб-сервере, что и файл класса апплета:

```
Properties props = new Properties();
try {
   URL url = new URL( getCodeBase(), "appletConfig.props" );
   props.load( url.openStream() );
} catch ( IOException e ) { /* ошибка*/ }
```

Более выгодный способ загрузить ресурсы — использовать методы `getResource()` и `getResourceAsStream()` класса `Class`, которые ищут файлы апплета JAR (если они есть) в его базе исходных кодов, являющейся дополнением директории classpath для апплетов. Следующий код загружает похожие данные более переносимым способом:

```
Properties props = new Properties();
try {
   props.load( getClass().getResourceAsStream( "appletConfig.props") );
} catch ( IOException e ) { /* ошибка */ }
```

Апплет может запросить программу просмотра вернуть изображение, вызвав метод `getImage()`. Расположение изображения будет получено из указанного абсолютного URL-адреса или извлечено из других ресурсов апплета:

```
public class MyApplet extends javax.swing.JApplet {
   public void init() {
   try {
         // абсолютный URL
         URL monaURL =
         new URL( "http://myserver/images/mona_lisa.gif");
         Image monaImage = getImage( monaURL );
         // URL ресурсов апплета
```

```
            URL daffyURL =
            getClass().getResource("cartoons/images/daffy.gif");
            Image daffyDuckImage = getImage( daffyURL );
        }
        catch ( MalformedURLException e ) {
            // неявный url
        }
    }
    // ...
}
```

Опять же использование метода `getResource()` предпочтительнее; он ищет изображение в JAR-файле апплета (если существует хотя бы один такой файл), перед тем как искать где-нибудь еще в директории classpath апплета на сервере. (Позже мы будем много говорить о классах, размещенных в апплетах.)

Управление браузером

Строка состояния — это текст, который обычно размещается где-либо в веб-браузере и сообщает о его текущей активности. Апплет может указать в запросе, что некий текст должен быть размещен в строке состояния, используя метод `showStatus()`. (Браузер не требует реагировать на данный запрос, но большинство браузеров обязывают выполнять его.)

Апплет также может направить запрос браузеру на отображение нового документа. Чтобы сделать это, апплет вызывает метод `showDocument(url)` класса `AppletContext`. Вы можете получить ссылку на объект класса `AppletContext`, используя метод апплета `getAppletContext()`. При вызове метода `showDocument(url)` заменяет текущий показанный документ, это означает, что запущенный вами апплет будет остановлен.

Другая версия метода `showDocument()` передает дополнительный аргумент `String`, чтобы сообщить браузеру, где отобразить новый URL:

```
getAppletContext().showDocument( url, name );
```

Аргумент `name` может быть именем существующего, именованного HTML-фрейма; документ, расположенный по указанному URL, будет отображен в данном фрейме. Вы можете использовать этот метод для создания апплетов, которые динамически направляют браузер к новым местоположениям, но хранить их активными на экране в отдель-

ном фрейме. Это типично для апплетов, которые выполняют функции панели навигации или меню. Если именованный фрейм не существует, браузер создает новое окно верхнего уровня для его хранения. Кроме того, аргумент name может иметь одно из следующих специальных значений:

`self`

Показывать в текущем фрейме

`_parent`

Показывать в родительском фрейме по отношению к текущему

`_top`

Показывать во фрейме верхнего уровня

`_blank`

Показывать в новом окне верхнего уровня

Запросы каждого из методов `showStatus()` и `showDocument()` могут быть проигнорированы безучастной программой просмотра или веб-браузером. Ничто не вечно в мире браузеров.

Внутреннее взаимодействие апплетов

Хотя это не общепринято, апплеты, которые встроены в документы, загруженные из того же местоположения на веб-сайте, могут использовать простой механизм, чтобы определять положение друг друга и координировать свою активность на странице. Когда апплет ссылается на другой апплет, он может взаимодействовать с ним, как с любым другим объектом, вызывая методы и отправляя события. Метод `getApplet()` просматривает содержимое апплета по его имени:

```
Applet clock = getAppletContext().getApplet("theClock");
```

Вы передаете имя апплета, используя в вашем документе HTML атрибут `name` в теге `<applet>`. Также вы можете использовать метод `getApplets()` для перечисления всех доступных на странице апплетов.

Сложность в отношении взаимодействия апплетов состоит в том, что они выполняются внутри «песочницы» безопасности. Недоверенный апплет может «видеть» и взаимодействовать только с объектами, которые были загружены одним и тем же загрузчиком класса. Сейчас единственным надежным критерием объединения загрузчиков классов

апплетов является использование общего основного URL. Например все апплеты, содержащиеся на веб-странице, загруженной с основного URL-адреса **http://foo.bar.com/mypages/**, должны использовать общий загрузчик классов и должны видеть друг друга. В данном случае будут включены документы *mypages/foo.html* и *mypages/bar.html*, но не документ *mypages/morestuff/foo.html*.

Когда апплеты используют общий загрузчик классов, можно использовать другую технику. Как и с любыми другими классами, вы можете вызывать статические методы апплета по имени. Другими словами, вы можете использовать статические методы в одном из ваших апплетов как реестр для координирования ваших действий.

Устойчивость и навигация апплетов

Одним из самых крупных недостатков Applet API является отсутствие надежного контекста для координации их активности в ходе навигации по многостраничному документу или веб-приложению. Applet API просто не приспособлена для этого. Хотя жизненный цикл апплета является вполне определенным с точки зрения данного API, но не является таковым с точки зрения управления браузером или сферы видимости. Как мы указали в предыдущей части, апплеты, загруженные из одной базы исходного кода, в среде исполнения могут быть обнаружены по их атрибутам имен. Однако нельзя точно определить, сколько просуществует апплет или когда остановится для разрушения или выйдет за пределы видимости. Если будете тестировать наш апплет `ShowApplet` в различных браузерах и, используя различные Java-плагины, которые мы обсудим позже, вы увидите, что в некоторых случаях апплет останавливается и запускается вновь, когда пользователь покидает страницу, но чаще апплет запускается с нуля. Это усложняет создание многостраничных приложений, способных корректно работать во всех браузерах.

Одно из решений: использовать статические методы, объединенные в «реестр», как упоминалось ранее. Однако управлять загрузкой классов апплетами еще сложнее, чем управлять самими апплетами. В Java 1.4 была добавлена пара методов в класс `AppletContext` для поддержки коротких терминов апплета: `setStream()` и `getStream()`. Используя эти методы, апплет может запрашивать контекст для сохранения потока байтовых данных по ключевым значениям и возвращения их позже. Предоставление состояния контекста как потока — немного странная идея, но к этому легко приспособиться. Далее пример:

```
getAppletContext.setStream("myStream",
   new ByteArrayInputStream( "Это некий тест данных...".getBytes() ) );
```

Позже потоковые данные могут быть запрошены:

```
InputStream in = getAppletContext.getStream( "myStream" );
```

В настоящее время, данные хранятся только пока запущен браузер. Если вам необходимы более сложное состояние и возможности навигации, рассмотрите использование записи в файл для подписанных апплетов или применение дополнений Java Web Start API для локальной установки вашего приложения.

Элемент `applet`

Апплеты встраиваются в HTML-документы тегом `<applet>`. Тег `<applet>` имеет сходство с HTML-тегом изображения `<img>`. Он содержит атрибуты, идентифицирующие апплет, который необходимо отобразить и, дополнительно, передающие веб-браузеру подсказки, как апплет должен быть отображен*.

Стандартный тег изображения имеет атрибуты размера и размещения, такие как `height` и `width`, их можно использовать и внутри тега апплета. Однако, в отличие от изображения, в апплете имеются оба тега: открывающий `<applet>` и завершающий `</applet>`. Между ними может быть заключено любое количество тегов `<param>`, которые содержат конфигурационные данные для передачи в апплет:

```
<applet attributeattribute ... >
   <param parameter >
   ...
</applet>
```

Атрибуты

Атрибуты — это пары имя/значение, которые интерпретируются веб-браузером или программой просмотра апплетов. Атрибуты тега `<applet>` определяют основные настройки, применяющиеся к любому

* Если вы не знакомы с языком разметки HTML, вы, возможно, захотите обратиться к книге «HTML и XHTML. Подробное руководство» Чака Маскиано и Билла Кеннеди («Символ плюс») для полного описания HTML и структуры веб-документов.

апплету, такие как размер и выравнивание. В определении тега `<applet>` перечислен фиксированный набор распознаваемых атрибутов; определение некорректного или несуществующего атрибута может привести к ошибке HTML.

В теге `<applet>` размещаются три атрибута. Два из них — `width` и `height` — определяют положение, занимаемое апплетом на экране. Третий необходимый атрибут должен быть или `code`, или `object`; вы должны использовать один из этих атрибутов и не можете указать оба атрибута одновременно. Атрибут `code` определяет файл класса, из которого загружается апплет; атрибут `object` определяет сериализуемое представление апплета. Наиболее часто вы будете использовать атрибут `code`.

Следующий фрагмент HTML для простого апплета с часами, который не получает параметров и не требует специальной разметки HTML:

```
<applet code="AnalogClock" width="100" height="100"></applet>
```

Файл HTML, который содержит тег `<applet>`, должен храниться в той же директории, что и файл класса *AnalogClock.class*. Элемент `applet` не чувствителен к пробелам в HTML-коде, поэтому предыдущий тег можно отформатировать в читаемом виде:

```
<applet
   code="AnalogClock"
   width="100"
   height="100">
</applet>
```

Параметры

Параметры — это аналог аргументов командной строки; они предоставляют способ для передачи данных апплету. Каждый тег `<param>` содержит атрибуты `name` и `value`, которые передают строковые значения апплету:

```
<param name = "parameter_name" value = "parameter_value">
```

Параметры обеспечивают средства внедрения специфичных данных приложения и конфигурационной информации внутри HTML-документа. Например, в нашем апплете `AnalogClock` должен быть доступен параметр, который выбирает между локальным и мировым часовым поясом:

```
<applet code="AnalogClock" width="100" height="100">
   <param name="zone" value="GMT">
</applet>
```

Предполагается, что апплет `AnalogClock` разработан так, что ищет в параметре `zone`, различные значения GMT (Greenwich Mean Time, среднее время по Гринвичу).

Имена и значения параметров должны быть понятными и могут содержать пробелы и другие символы разрывов. Параметры передаются апплету, за исключением, конечно, определенных разработчиком данного апплета. Не существует стандартов присваивания имен или значений, апплет сам интерпретирует пары имя/значение параметров, которые переданы ему. Может быть задано любое количество параметров, и апплет может использовать или игнорировать их, как сочтет нужным.

¿Habla Applet?

Веб-браузеры игнорируют теги, которые не понимают; если веб-браузер не знает тегов `<applet>` и `<param>`, мы должны быть готовы к их исчезновению и тому, что любой HTML-код между тегами `<applet>` и `</applet>` отобразится в браузере. Обычно веб-браузеры с включенной поддержкой Java игнорируют любой дополнительный HTML между тегами `<applet>` и `</applet>`. Все это означает, что мы можем разместить некоторый замещающий HTML текст внутри тега `<applet>`, который будет отображаться только веб-браузерами, не способными выполнить апплет. Для нашего примера `AnalogClock`, мы могли бы отобразить небольшой текст описания и изображение апплета часов в качестве рекламы:

```
<applet code="AnalogClock" width="100" height="100">
   <param name="zone" value="GMT">
   <strong> Если вы видите ЭТО, ваш веб-браузер не поддерживает Java. Это
изображение того, что вы упускаете. </strong>
   <img src="clockface.gif">
</applet>
```

Полная версия тега `<applet>`

Разберем подробно синтаксис тега `<applet>`:

```
<applet
   code = имя_класса
```

или:

```
   object = сериализованное_имя_апплета
   width = высота_в_пикселах
   height = ширина_в_пикселах
   [ codebase = URL_исходных_файлов ]
   [ archive = разделенный_запятыми_список_архивных_файлов ]
   [ name = имя_экземпляра_апплета ]
   [ alt = замещающий_текст ]
   [ align = стиль ]
   [ vspace = вертикальный отступ в пикселах ]
   [ hspace = горизонтальный отступ в пикселах ]
>
   [ <param name = имя_параметра value = значение_параметра> ]
   [ <param ... ]
   [ HTML-код для браузеров не поддерживающих Java ]
</applet>
```

Атрибут `code` или атрибут `object` должны присутствовать для указания выполняемого апплета. Атрибут `code` встречается чаще и определяет файл класса апплета. Атрибут `object` определяет сериализованное представление апплета. Когда вы используете атрибут `object` для загрузки апплета, метод апплета `init()` не вызывается. Вместо этого вызывается метод апплета `start()`.

Атрибуты `width`, `height`, `align`, `vspace`, и `hspace` определяют предпочтительные размер, выравнивание и отступ соответственно. Атрибуты `width` и `height` обязательны.

Атрибут `codebase` определяет основной URL для поиска файла класса апплета. Если этот атрибут не указан, браузер выполняет поиск в той же директории, где размещен файл HTML. Атрибут `archive` определяет список файлов JAR или ZIP, в которых размещены файлы класса апплета. Чтобы разместить два или более файлов в списке, разделите их запятыми; например следующий атрибут указывает браузеру найти и загрузить три архива классов апплета:

```
archive="Part1.jar,Part2.jar,Utilities.jar"
```

Файлы архива, перечисленные в атрибуте `archive`, загружаются с URL, указанного в атрибуте `codebase`. При поиске классов браузер проверяет архивы прежде, чем выполнять поиск в других местах на сервере.

Атрибут `alt` определяет альтернативный текст отображаемый браузером, который понимает тег `<applet>` и его атрибуты, но в котором отключена поддержка Java или не выполняется апплет. Этот атрибут может также описывать апплет, поскольку в этом случае любой HTML-код, размещенный между тегами `<applet>` и `</applet>`, при конверсии будет проигнорирован браузерами, в которых включена поддержка Java.

Атрибут `name` определяет имя экземпляра исполняемого апплета. Это имя определяется как уникальный ярлык для каждой копии на конкретной HTML странице. Например, если мы включаем наши часы дважды в одну страницу (используем два тега апплета), мы должны присвоить каждому экземпляру уникальное имя, чтобы различать их:

```
<applet code="AnalogClock" name="bigClock" width="300" height="300">
</applet>
<applet code="AnalogClock" name="smallClock" width="50" height="50">
</applet>
```

Апплеты могут использовать имена экземпляров для распознавания и взаимодействия с другими апплетами на одной странице. Мы могли создать апплет «clock setter» для настройки времени в апплете `AnalogClock` и передать имя его экземпляра частной цели на этой странице, как параметр. Это должно выглядеть приблизительно так:

```
<applet code="ClockSetter">
   <param name="clockToSet" value="bigClock">
</applet>
```

Загрузка файлов классов

Атрибут `code` тега `<applet>` должен указывать имя апплета. Это может быть просто имя класса либо путь к пакету и имя класса. Сейчас давайте рассмотрим случай, когда указывается только имя класса, пакеты мы обсудим немного позже. По умолчанию система исполнения Java ищет файл класса в той же директории, где размещен HTML-документ. Это расположение также известно как основной URL-адрес гипертекстового документа.

Рассмотрим HTML-документ *clock.html*, содержащий пример нашего апплета с часами:

```
<applet code="AnalogClock" width="100" height="100"></applet>
```

Предположим, что мы извлекаем документ со следующего URL:

```
http://www.time.ch/documents/clock.html
```

Java пытается извлечь файл класса апплета из той же директории:

```
http://www.time.ch/documents/AnalogClock.class
```

Атрибут `codebase` тега `<applet>` указывает альтернативный основной URL для поиска файла класса. Давайте теперь определим атрибут `codebase` в нашем HTML-документе, как в следующем примере:

```
<applet
   codebase="http://www.joes.ch/stuff/"
   code="AnalogClock"
   width="100"
   height="100">
</applet>
```

Теперь Java ищет файл класса апплета в директории:

```
http://www.joes.ch/stuff/AnalogClock.class
```

Пакеты

Для «разбросанных» файлов классов апплета, не собранных в архивы, Java использует стандартное имя пакета для указания пути к директории, где размещены файлы на сервере. Разница лишь в том, что вместо локального выполнения запроса поиска файлов выполняется запрос поиска на веб-сервере по основным URL-адресам исходных файлов апплета. Перед извлечением файла класса с сервера, имя его пакета передается клиенту с указанием относительного пути от основной директории расположения исходных файлов апплета.

Предположим, что наш апплет `AnalogClock` размещен в пакете с названием `time.clock` (подчиненный пакет для классов работы с часами, внутри пакета классов, работающих со временем). Полное имя нашего класса — `time.clock.AnalogClock`. Тогда наш простой тег `<applet>` выглядел бы так:

```
<applet code="time.clock.AnalogClock" width="100" height="100"></applet>
```

Опять же скажем, что документ *clock.html* получен из:

```
http://www.time.ch/documents/clock.html
```

Теперь Java ищет файл класса в следующей директории:

```
http://www.time.ch/documents/time/clock/AnalogClock.class
```

Аналогично, когда указывается альтернативный атрибут `codebase`:

```
<applet
   codebase="http://www.joes.ch/stuff/"
   code="time.clock.AnalogClock"
   width="100"
   height="100">
</applet>
```

Теперь Java пытается найти класс по соответствующему пути относительно указанного основного URL:

```
http://www.joes.ch/stuff/time/clock/AnalogClock.class
```

appletviewer

Java SDK (Java Development Kit) — свободно распространяемый комплект разработчика приложений на языке Java, поставляется с программой просмотра апплетов как нельзя лучше названой `appletviewer`. Для использования `appletviewer` укажите URL-адрес документа в командной строке. К слову для просмотра, пока только теоретически, нашего примера `AnalogClock` с указанным ранее URL-адресом используйте следующую команду:

```
% appletviewer http://www.time.ch/documents/clock.html
```

Программа `appletviewer` извлекает все апплеты из указанного документа и отображает каждый в отдельном окне. `Appletviewer` не является веб-браузером и не пытается отображать HTML.

Изначально, до появления плагина Java, эта программа была полезна для тестирования апплета в отдельных версиях среды выполнения Java.

Java Web Start

Технология Java Web Start API — это альтернатива применению апплетов. Java Web Start использует протокол Java Network Launching

Protocol (JNLP) для прозрачной загрузки и локальной установки приложений java. Единственное, что необходимо сделать пользователю — это перейти по ссылке установки на веб-странице. Впоследствии установленные приложения могут быть запущены как любое другое приложение, нажатием на ярлык на рабочем столе или через меню Пуск, но при этом они продолжают управляться политикой безопасности Java, если иное не разрешено пользователем. Приложения Web Start также автоматически проверяют обновления и самостоятельно обновляются через сеть. Java Web Start — это *не требующий администрирования* вид клиентской установки. Это подразумевает, что клиент не выполняет никаких действий для установки или обслуживания приложения. JNLP приложения могут быть подписаными (позволяя пользователю предоставлять им высоко детализированные привилегии) или неподписанными. Но даже неподписанные JNLP приложения могут воспользоваться стандартными API, которые запрашивают у пользователя разрешение на выполнение таких основных операций, как открытие файлов и печать.

Создать пакет вашего приложения использующего протокол JNLP относительно просто, и мы не будем рассматривать этого здесь. Процесс в основном включает создание JNLP-файла, который содержит список ваших JAR-файлов и определяет специальные полномочия, необходимые им. Далее вы должны добавить соответствующие ссылки на вашей веб-странице использующей Web Start. При первой попытке пользователя установить JNLP приложение ему потребуется установить Java плагин. Впоследствии, компонент Java Web Start управляет всеми JNLP установками. Посетите веб-сайт Oracle для получения дополнительной информации.

Заключение

В этой главе мы осветили некоторые события, которые привели к нынешнему несовершенному положению Java в веб-браузерах, и в общих чертах описали текущую ситуацию. Java-плагин в настоящее время — ваша единственная функция для запуска Java в браузере. Новые технологии, такие как Java Web Start, предоставляют альтернативные направления разработки клиентских приложений, которые также могут быть применены. Кроме того, существуют различные сторонние программы установки Java-приложений, управляемые мышью, которые вы, возможно, захотите рассмотреть.

Глава 24
XML

Время от времени появляется идея, которая кажется настолько простой и очевидной, что каждый задается вопросом: почему этого не было ясно с самого начала. При этом часто оказывается, что идея не так уж и нова. Революция Java началась с идей, основанных на языках программирования, появившихся ранее.

XML — Extensible Markup Language, расширяемый язык разметки, сделал для контента то же, что язык Java сделал для программирования: опираясь на некоторые старые идеи, использовал их, чтобы предоставить переносимый способ описания данных.

XML — это простой, распространенный формат для представления структурной информации в текстовом виде. Концепция XML следует успеху HTML, как универсального формата представления документов и обобщает его для обработки различных видов данных. В процессе XML не только изменил HTML, но трансформировал способ мышления многих предприятий об их информации. XML важная фундаментальная технология для мира, в котором все большую роль играет обмен документами и данными.

Обзор

Эта глава одна из самых больших в книге и описывает множество API и концепций. Отчасти это связано с произошедшей эволюцией XML инструментов в части поддержки работы с XML на различных уровнях абстракции. В этой главе мы собираемся представить вам API, которые, как мы думаем, остаются важными и полезными. Начав с нижнего уровня, мы пройдем весь путь наверх. Сначала мы рассмотрим основные XML концепции и низкоуровневые API: управляющий событиями SAX (Simple API for Java) и создающий модель DOM (Document Object Model). Также мы обсудим родственные технологии, такие как верификация XML Schema, запросы XPath, и преобразование XSL (Extensible Style Sheet). Далее в этой главе мы обсудим высокоуровневые JAXB,

Java XML Binding API, для размещения простых Java-объектов непосредственно в XML и обратных действий.

Это означает, что для некоторых из вас наиболее полезный материал расположен в конце этой главы, где мы описываем высокоуровневые инструменты. Так что хотим вас заверить: чем дальше, тем будет интереснее. Когда мы дойдем до части JAXB, мы увидим, что можем взять простые объекты Java (POJO) и записать их в XML, добавляя (в самом простом случае) однострочное описание. Следующий фрагмент показывает Java-классы `Person` и `Address` и соответствующий XML, в котором они будут автоматически размещены.

```
@XmlRootElement
public class Person {
public String name;
public Address address;
public int age;
}
public class Address {
public String city, street;
public int number, zip;
}
<person>
<name>Пэт Нимейер</name>
<address>
<city>Сент-Луис</city>
<street>улица Java</street>
<number>1234</number>
<zip>54321</zip>
</address>
<age>42</age>
</person>
```

Прежде чем мы перейдем к этому, давайте сделаем шаг назад и поговорим о стимулах и «правилах» XML и некоторых способах, которыми мы можем выполнять их синтаксический анализ и создавать их.

Немного предыстории

XML и HTML называются *языками разметки* в соответствии с тем, как они добавляют структуру в обычный текстовый документ: обрамле-

ние участков текста тегами, указывающими структурный элемент или значение, похоже на выделение предложения цветной ручкой или добавление заметки. В то время как в HTML предопределен набор тегов и их структура, XML подобен чистому листу, в котором автор сам определяет теги, правила и их значения.

И XML, и HTML обязаны своим происхождением Standard Generalized Markup Language (SGML) — прародителю всех языков разметки. SGML годами использовался в индустрии печати (включая издательство O'Reilly). Но это было лишь до того, как Веб захватил мир и стал основной сферой применения HTML. HTML начинался как очень небольшое приложение SGML, и если HTML и сделал что-то, то он доказал, что правит простота.

Сравнение текстовых и двоичных данных

Когда Тим Бернерс-Ли (Tim Berners-Lee) начал создавать постулаты Всемирной паутины в ЦЕРН в конце 1980-х, он хотел организовать проектную информацию, используя гипертекст со ссылками, встроенными в обычный текст*. Когда Всемирной паутине потребовался протокол, был изобретен HTTP — простой клиент-серверный протокол на основе текста. Так чем же действительно столь привлекательна идея простого текста? Почему, например, Тим не использовал формат Microsoft Word как основу веб-документов? Наверное, протокол двоичного формата, не предназначенный для человеческого восприятия и столь же машинно-ориентированный, мог быть эффективнее? С момента создания Веб в нем произошли буквально триллионы HTTP-транзакций. Была ли это действительно хорошая идея — использовать английские слова «GET» (получить) и «POST» (отправить) как часть протокола?

Как мы все видим, ответ однозначно да! С тем, что может прочитать и написать человек, человеку-разработчику проще работать. Для высокого уровня оптимизации (и прозрачности) существуют свои время и место, но, когда целью является повсеместное признание и кроссплатформенная независимость, простота и прозрачность имеют первостепенное значение. Итак, первое фундаментальное положение XML: простота и номинальная удобочитаемость данных.

---

* Чтобы прочитать первоначальное предложение Бернерс-Ли, посетите страницу *www.w3.org/History/1989/proposal.html*.

Универсальный синтаксический анализатор

Использовать текст для обмена данными на самом деле не новая идея, но так исторически сложилось, что вместе с появлением каждого нового формата документов появлялся новый синтаксический анализатор. Синтаксический анализатор — это приложение, которое считывает документ и понимает принятый в нем формат, обычно применяя некоторые правила к его содержимому. Например, Java-класс `Properties` содержит синтаксический анализатор для стандартного формата файла свойств (глава 11). В нашей простой таблице (глава 18), мы создали синтаксический анализатор, понимающий основные математические выражения. Как мы увидели, в зависимости от сложности синтаксический анализ может быть довольно замысловатым. В XML мы можем представлять данные без необходимости писать такого рода распространенные синтаксические анализаторы. Это не значит, что разумно использовать XML для всего (например, для записи математических выражений в нашей таблице), но для обычных типов информации, которыми мы обмениваемся в Интернете, нам нет необходимости писать синтаксический анализатор для работы с основным синтаксисом и управлением строками. Совместно с компонентами проверки документов (Document Type Definitions [DTDs] или XML Schema) автоматически выполняется проверка большинства ошибок. Итак, второе фундаментальное положение XML: стандартизованный синтаксический анализ и верификация.

Положение языка XML

В этой главе мы обсуждаем мощные и популярные API. Они ежедневно используются по всему миру для создания систем в производственных масштабах. В предыдущие годы, JAXB для связки Java и XML значительно оптимизировал и упростил (в основном за счет использования Java-аннотаций для замены конфигурационных файлов и поддержки стратегии «code first»).

Однако, как и со всеми популярными технологиями, имело место признание ее ограничений и некоторой сложности, закравшейся в то, что начиналось с простой концепции. В области браузерных приложений, некоторые обратились к JavaScript Object Notation (JSON) как к даже более легкому способу, который изначально представлен в JavaScript, особенно для переходных клиент-серверных связей. Однако XML инструменты все еще широко используются в этой области с тем же успехом. Protocol Buffers-encoding — схема разработанная Google, еще один

пример связи «система — система», который используется вместо XML. В этом случае гибкость является основным преимуществом высокой производительности. Но XML — наиболее мощный обобщенный формат документов и обмена данными с самым широким спектром инструментов поддержки.

XML API

Все основные API для работы с XML сейчас входят в стандартный релиз Java. Включая стандартный пакет расширений *javax.xml* для работы с Simple API for XML (SAX), Document `Object` Model (DOM), XML Binding JAXB и Extensible Stylesheet Language (XSL) преобразования, так же как и API вроде XPath и XInclude. Если вы используете старую версию Java, вы все еще можете использовать большинство из этих инструментов, но вам придется отдельно загрузить данные пакеты.

XML и веб-браузеры

Современные веб-браузеры полностью поддерживают XML, как в виде простой визуализации XML-контента, так и трансформации XML в HTML на клиентской стороне для отображения. Если вы загрузите XML-документ в ваш браузер, в основном он будет отображен как дерево с возможностью управлять сворачиванием и раскрытием узлов (как схем). Отображение XML таким способом используется в основном для отладки, но JavaScript может также поддерживать преобразование XSL на клиентской стороне напрямую в браузере. XSL — это язык для преобразования XML в другие документы; мы расскажем об этом далее в этой главе.

При просмотре в более старых браузерах или в контексте не вполне явного XML-формата браузер будет в основном просто отображать текст документа без всех тегов (структурной информации). Таково предписанное поведение для работы с неизвестной разметкой XML в среде просмотра. Запомните, что вы можете использовать команду просмотра исходного кода страницы для отображения текста файла в вашем браузере, если вы хотите увидеть оригинал исходного файла.

Основы XML

Основа синтаксиса XML крайне проста. Если вы работали с HTML, считайте, что вы уже наполовину знаете его. Как и HTML, XML представ-

ляет информацию о тексте, используя *теги* для формирования структуры. Тег начинается с имени, заключенного между символами (<) «меньше» и (>) «больше». В отличие от HTML, XML-теги всегда должны быть *парными*, иначе говоря, за открывающим тегом всегда должен следовать закрывающий тег. Он выглядит в точности как и открывающий, однако начинается с символов (</) «меньше» и «слеш». Открывающий тег, закрывающий тег и любой контент между ними совместно представляют *элемент* XML документа. Элементы могут содержать другие элементы, но они должны быть правильно вложены (все теги открытые внутри элемента должны закрыться внутри него). Элементы также могут содержать простой текст или смесь элементов и текста (которая называется смешанный контекст). Комментарии размещаются между маркерами <!-- и -->. Несколько примеров приведено ниже:

```
<!-- Просто -->
<Sentence>Здесь размещен текст.</ Sentence>
<!-- Элемент -->
<Paragraph><Sentence>Здесь размещен текст</Sentence></Paragraph>
<!-- Смешанный -->
<Paragraph>
<Sentence>Здесь <глагол> размещен </глагол> текст.</Sentence>
</Paragraph>
<!-- Empty -->
<PageBreak></PageBreak>
```

Пустой тег может быть записан более компактно в специальной форме, с однократным использованием завершающего тега с символами слеш и больше (/>):

```
<PageBreak/>
```

Атрибуты

XML-элемент может содержать *атрибуты*, которые представляют собой простые пары имя — значение, размещенные внутри открывающего тега:

```
<Document type="LEGAL"id="42">...</Document>
<Image name="truffle.jpg"/>
```

Значение атрибута всегда должно быть заключено в кавычки. Вы можете использовать двойные (") или одиночные (') кавычки. Оди-

ночные кавычки полезны, если значение содержит двойные кавычки. Атрибуты предназначены для использования простых, неструктурированных свойств или компактных идентификаторов связанных с данными элемента. Всегда возможно обратить атрибут в дочерний элемент, строго говоря, реальной необходимости в атрибутах нет. Но часто они делают XML более понятным и логичным. В случае элемента `Document` в нашем предыдущем фрагменте атрибуты `type` и `ID` представляют метаданные документа. Мы должны ожидать, что Java-класс, представляющий `Document`, должен перечислить типы документов, включая `LEGAL`. В случае элемента `Image` атрибут, всего лишь более компактный способ подключения файла. Как правило, атрибуты должны быть компактными, с небольшой значащей структурой (URL выходят за границы); напротив, дочерние элементы могут быть крайне сложными.

Атрибут `id` в предыдущем примере может иметь особое значение, когда используется с соответствующим атрибутом `idref`. Совместно эти стандартные атрибуты используются при верификации документа для обеспечения его целостности. При верификации значение атрибута `id` должно быть уникальным внутри документа и значение атрибута `idref` должно обращаться к верному `id` в документе.

XML-документы

XML-документ начинается с заголовка подобного следующему и одного *корневого элемента:*

```
<?xml version="1.0" encoding="UTF-8"?>
<MyDocument>
</MyDocument>
```

Заголовок определяет версию XML и используемую кодировку. Корневой элемент — это просто верхний элемент в иерархии, которую можно представить в виде дерева. Если вы пренебрегаете заголовком или используете XML-текст не с единственным корневым элементом, как в наших простых предыдущих примерах, то формально это будет называться XML-*фрагмент*.

Кодировка

По умолчанию в XML-документе используется кодировка UTF-8, 8-битная ASCII кодировка Unicode. Эта кодировка сохраняет значения

ASCII, поэтому английский текст не изменяется. Она также поддерживает хранение значений Unicode достаточно эффективным способом. В XML-документе может быть определено использование другой кодировки в атрибуте XML-заголовка. Внутри XML-документа некоторые символы неприкосновенны, например символы < и >, обозначающие теги элементов. Когда вам необходимо включить их в ваш текст, вы должны закодировать их. XML предоставляет исключающий механизм, называемый «сущности», который позволяет кодировать специальные структуры. XML имеет пять предопределенных сущностей, они показаны в табл 24.1.

Табл. 24.1. XML-сущности

Сущность	Кодировка
`&`	& (амперсанд)
`<`	< (меньше)
`>`	> (больше)
`"`	" (кавычки)
`'`	' (апостроф)

Альтернативный способ подобной кодировки текста заключается в использовании специальной «синтаксически не анализируемой» секции текста, называемой *секцией символьных данных* (character data, CDATA). Секция CDATA начинается со скрытой строки `<![CDATA[` и заканчивается `]]>`, как эта:

```
<![CDATA[ Learning Java, O'Reilly & Associates ]]>
```

Секция CDATA выглядит как комментарий, но все еще остается частью документа, только нераспознаваемой синтаксическим анализатором. Существует еще один альтернативный способ, который использует специальную директиву `<include>` для добавления содержимого `URL` или `file` либо как предварительно избегаемого текста, либо не обязательно синтаксически анализируемого XML. Включения XML очень удобны, и мы поговорим о них позже в этой главе.

Пространство имен

Возможно, вы заметили, что в HTML используется тег `<body>` для создания структуры веб-страниц. Представьте на мгновение, что мы пишем XML-код для похоронного бюро, где также используется тег `<body>`,

но для другой, более мрачной цели. Это может оказаться проблемой, если мы захотим смешать HTML с информацией о морге.

Если вы сравните теги HTML и похоронного бюро, в данном случае как языки, элементы (имена тегов), используемые в документе, — действительно лексика этих языков. *Пространство имен XML* — это способ сообщить, какой словарь вы используете для данного элемента, что позволяет нам свободно смешивать их. Позже мы расскажем о XML-схемах, которые соблюдают грамматику и синтаксис языка.

Пространство имен определяется атрибутом `xmlns`, чье значение — это уникальный идентификатор ресурса (Uniform Resource Identifier, URI), который однозначно определяет набор и, как правило, значение тегов из этого пространства имен:

```
<element xmlns="namespaceURI">
```

Напомним, как вы узнали из главы 14, URI не обязательно является URL. URI более обобщенные, чем URL. С практической точки зрения, URI должен рассматриваться как уникальная строка. Часто URI фактически является URL-адресом документа, описывающего пространство имен, но когда это верно только по соглашению.

Атрибут пространства имен `xmlns` может быть применен к элементу и будет действовать на все его дочерние (вложенные) элементы, что называется пространством имен по умолчанию для элемента:

```
<body xmlns="http://funeral-procedures.org/">
```

Часто желательно смешивать и сочетать пространства имен тег за тегом. Для этого мы можем использовать специальный атрибут, определяя специальный идентификатор пространства имен и используя этот идентификатор как префикс тегов. Например:

```
<funeral xmlns:fun="http://funeral-procedures.org/">
<html><head></head><body>
<fun:body>Корпус #42</fun:body>
</funeral>
```

В предыдущем фрагменте XML мы обозначили тег `body` префиксом `"fun:"`, который мы определили в теге `<funeral>`. В этом случае мы также должны используя рефлексию обозначить корневой тег:

```
<fun:funeral xmlns:fun="http://funeral-procedures.org/">
```

Фабрики синтаксических анализаторов XML, поставляемые вместе с Java, имеют переключатель, который определяет, хотите ли вы использовать синтаксический анализатор для интерпретации пространства имен. По историческим причинам этот переключатель по умолчанию отключен.

```
parserFactory.setNamespaceAware( true );
```

Мы расскажем подробнее о синтаксическом анализе в разделах, посвященных SAX и DOM, далее в этой главе.

Проверка

Документ, соответствующий основным правилам XML в верной кодировке и сбалансированными тегами, называется *синтаксически верным* документом. Это означает, что документ синтаксически корректен, однако не несет никакого другого смысла. Два связанных набора инструментов DTD и XML Schemas определяют способы предоставления грамматики ваших XML-элементов. Они позволяют вам создавать синтаксические правила, такие как «элемент `City` может появиться только один раз в элементе `Address` и перед элементом `State`». XML Schema пошел еще дальше, предоставляя гибкий язык для описания правильности обрабатываемого контента тегов, включая и простой, и сложный типы данных, созданные из чисел и строк. Документ, который проверяется на соответствие DTD или описанию XML Schema и следующий правилам, называется *верифицированным* документом. Файл может быть синтаксически верным и при этом не верифицированным, но не наоборот.

HTML в XHTML

Говоря очень кратко, мы могли бы сказать, что наиболее популярной и широко используемой формой XML в мире сегодня является HTML. Но это будет неточное употребление термина, потому что HTML — это на самом деле синтаксически неверный XML. Теги HTML нарушают правило XML, запрещающее наличие несбалансированных элементов, например распространенный тег `<p>` обычно используется без завершающего тега. Атрибуты HTML также не требуют кавычек. Теги XML учитывают регистр: `<P>` и `<p>` — это два разных тега в XML. В целом мы могли сказать, что HTML «снисходителен» в отношении подобных деталей, но как разработчик вы знаете, что свободный синтаксис приводит

к двусмысленности. XHTML — альтернативная, строгая XML-версия HTML, которая понятна и недвусмысленна. Эта форма HTML работает в современных браузерах. К счастью, если вы хотите переключиться на нее, вы не должны вручную подчищать все ваши HTML-документы; программа с открытым исходным кодом Tidy автоматически конвертирует HTML в XHTML, верифицирует его и исправляет распространенные ошибки.

SAX

SAX — это низкоуровневый, событийный API, предназначенный для синтаксического анализа XML-документов. SAX появился в Java, но может быть реализован во многих языках. Мы начнем наше обсуждение Java XML API с низкого уровня, постепенно поднимаясь до API высокого уровня (зачастую более удобных).

SAX API

Для использования SAX мы будем опираться на классы пакета `org.xml.sax`, стандартизированного W3C. Этот пакет содержит интерфейсы, распространенные во всех реализациях SAX. Для выполнения синтаксического анализа нам потребуется стандартный Java-пакет `javax.xml.parsers` для доступа к синтаксическим анализаторам XML. Пакет `java.xml.parsers` — это часть Java API для обработки XML (Java API for XML Processing, JAXP), который поддерживает различные реализации синтаксических анализаторов для использования в Java портативным способом.

Для чтения XML документа с применением SAX мы сначала регистрируем класс `org.xml.sax.ContentHandler` в синтаксическом анализаторе. Класс `ContentHandler` содержит методы, которые вызываются в ответ на части документа. Например, метод `startElement()` класса `ContentHandler` вызывается, когда обнаружен открывающий тег, а метод `endElement()` вызывается, когда найден закрывающий тег. Атрибуты обрабатываются вызовом метода `startElement()`. Текстовый контент элемента передается отдельным методом, называемым `characters()`. Метод `characters()` может быть вызван повторно для поставки большего количества текста, по мере чтения, но он всегда получает всю строку за раз. Ниже приведен способ описания этих методов в классе `ContentHandler`.

```
public void startElement(
String namespace, String localname, String qname, Attributes atts );
public void characters(
char[] ch, int start, int len );
public void endElement(
String namespace, String localname, String qname );
```

Параметр qname — это *полное имя* элемента: имя элемента с префиксом любого пространства имен, которое может быть применено. Когда вы работаете с пространством имен, параметры namespace и localname также передаются, раздельно представляя пространство имен и неполное имя элемента.

Интерфейс ContentHandler также содержит методы, вызываемые для начала и конца документа: startDocument() и endDocument(), а также специальные инструкции XML для обработки сопоставления пространства имен, и пробелов, которые не являются частью текстового контента и могут быть проигнорированы. В наших примерах мы ограничимся тремя предыдущими методами. Как и в случае большинства других интерфейсов Java, предоставляется простая реализация org.xml.sax.helpers.DefaultHandler, что позволяет нам переопределять только те методы, в которых мы заинтересованы.

JAXP

Для выполнения синтаксического анализа нам потребуется синтаксический анализатор из пакета javax.xml.parsers. JAXP абстрагирует процесс получения синтаксического анализатора через *фабрику шаблонов проектирования*, позволяя присоединять различные реализации синтаксических анализаторов к платформе Java. Следующий фрагмент создает объект SAXParser и затем получает объект XMLReader для использования в структурном анализе файла:

```
import javax.xml.parsers.*;
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
XMLReader reader = saxParser.getXMLReader();
reader.setContentHandler( myContentHandler );
reader.parse( "myfile.xml" );
```

Должно быть, вы ожидали, что объект SAXParser содержит метод parse. Класс посредник XMLReader был добавлен в SAX API в изменениях версий между 1.0 и 2.0. Позже мы обсудим некоторые возмож-

ности, которые могут быть настроены для управления действиями синтаксического анализатора XML. Эти возможности обычно настраиваются через методы фабрики синтаксического анализатора (например, `SAXParserFactory`) и сами не являются структурными анализаторами. Это обусловлено тем, что фабрики могут использовать различные реализации для поддержки различных требуемых особенностей.

Достоинства и недостатки SAX

Основная причина использовать SAX вместо API высокого уровня, которые мы будем обсуждать позже, заключается в том, что он легкий и событийно-ориентированный. SAX не требует хранения всего документа в памяти. Так, например, если вам необходимо получить текст лишь нескольких элементов документа или если вам необходимо достать элементы из большого потока XML, используя SAX, вы можете выполнить это эффективно. Событийная ориентированность SAX также позволяет вам выполнять действия, когда синтаксически анализируются начальный и конечный тег. Это может пригодиться для непосредственного управления вашими собственными моделями без прохождения через байт-представления. Главная слабость SAX в том, что вы действуете на уровне тегов без помощи синтаксического анализатора для хранения контекста.

Мы расскажем о том, как обойти это ограничение далее. Позже мы также расскажем о новом XPath API, который сочетает в себе большинство преимуществ SAX и DOM в более простой в использовании форме.

Создание модели с использованием SAX

Механизм получения SAX событий `ContentHandler` очень прост. Достаточно легко понять, как использовать его для получения значений или атрибутов отдельного элемента в документе. Что сложнее, так это понять, как использовать SAX для наполнения реальной объектной модели Java. Создание или размещение данных в Java-объектах из XML — настолько распространенная деятельность, что стоит рассмотреть SAX API применительно к этой проблеме. Следующий пример, `SAXModelBuilder`, делает именно это, считывает XML-описание и создает Java-объекты по команде. Этот пример немного необычен тем, что мы прибегаем к использованию рефлексии для выполнения этой задачи, но в этом случае мы пробуем взаимодействовать с Java-объектами динамически.

В этой части мы начнем с создания XML вместе с соответствующими классами Java, которые служат в качестве модели этого XML. Код данного примера сфокусирован на создании базового конструктора модели, который использует SAX для чтения XML и наполняет классы модели их данными. Идея в том, что разработчик создает только XML и классы модели, никакого самописного кода для выполнения синтаксического анализа. Вы могли бы использовать подобный код для чтения конфигурационных файлов приложений или для реализации нестандартного «языка» XML для описания рабочих процессов.

Преимущество в том, что в приложении нет настоящего кода синтаксического анализа, только в характерных инструментах конструктора. Наконец, далее в этой главе при обсуждении более мощных JAXB API мы повторно используем объектную Java-модель из этого примера, просто добавив несколько описаний.

Создание XML-файла

Первое, что нам понадобится — хороший XML-документ для синтаксического анализа. К счастью, пришло время инвентаризации в зоопарке! Следующий документ, *zooinventory.xml*, описывает двух обитателей зоопарка, включая некоторую жизненно важную информацию об их питании:

```
<?xml version="1.0" encoding="UTF-8"?>
<inventory>
<animal animalClass="млекопитающее">
<name>Сон Фон</name>
<species>Большая панда</species>
<habitat>Китай</habitat>
<food>Бамбук</food>
<temperament>Дружелюбный</temperament>
<weight>45.0</weight>
</animal>
<animal animalClass="млекопитающее">
<name>Какао</name>
<species>Горилла</species>
<habitat>Центральная Африка</habitat>
<foodRecipe>
<name>Горилла Чоу</name>
<ingredient>Фрукты</ingredient>
<ingredient>Ростки</ingredient>
```

```
<ingredient>Листья</ingredient>
</foodRecipe>
<temperament>Всезнайка</temperament>
<weight>45.0</weight>
</animal>
</inventory>
```

Документ невероятно прост. Корневой элемент <inventory> содержит два дочерних элемента <animal>. Элемент <animal> содержит несколько простых текстовых элементов, для таких вещей, как имя, вид и место обитания. Он также содержит или простой элемент <food>, или сложный элемент <foodRecipe>. Наконец, отметим, что элемент <animal> содержит один атрибут: animalClass, который описывает зоологическую принадлежность существ (например, млекопитающее, птица, рыба и т. д.). Это позволяет нам представить набор особенностей XML для работы с нашим примером.

Модель

Сейчас давайте создадим объектную Java-модель для инвентаризации в зоопарке. Эта часть очень рутинная: мы просто создаем класс для каждого из сложных типов элементов в нашем XML. Все прочее — просто строка или число. Пожалуй, лучше всего будет использовать стандартное свойство шаблона проектирования JavaBeans — то есть использовать частные поля (переменные экземпляра) с парой геттера и сеттера для каждого свойства. Однако, так как эти классы просто содержат данные и мы хотим, чтобы наш пример остался небольшим, мы выберем для использования публичные поля. Все, что мы собираемся сделать в этом примере, и, что более важно, все, что мы собираемся сделать, когда повторно используем эту модель далее в примере JAXB связки, может быть сделано для работы или с полями, или эквивалентными свойствами JavaBeans на основе методов. В этом примере важно лишь то, как мы устанавливаем переменные, а в дальнейшем, в случае JAXB, будет важно, где мы размещаем аннотации. Соответственно, классы:

```
public class Inventory {
public List<Animal> animal = new ArrayList<>();
}
public class Animal
{
public static enum AnimalClass { млекопитающее, рептилия, птица, рыба,
амфибия,беспозвоночное }
```

```
public AnimalClass animalClass;
public String name, species, habitat, food, temperament;
public Double weight;
public FoodRecipe foodRecipe;
public String toString() { return name +"("+animalClass+",
"+species+")"; }
}
public class FoodRecipe
{
public String name;
public List<String> ingredient = new ArrayList<String>();
public String toString() { return name + ": "+ ingredient.toString(); }
}
```

Как вы можете видеть, в случаях, где нам требуется представить последовательность элементов (например, элементы `animal` в элементе `inventory`), мы должны использовать коллекцию `List`. Также заметим, что свойство, которое будет служить для хранения нашего атрибута `animalClass` (например, млекопитающее), представлено типом `enum`. Мы также отправляем его в простой метод `toString()` для последующего использования. И еще: мы используем форму единственного числа для названий наших коллекций, например «animal», а не «animals», только потому, что это удобно. Мы расскажем об отображении имен больше в примере JAXB.

SAXModelBuilder

Вернемся к делу и напишем наш конструктор. Теперь мы можем сделать это, используя SAX API в сочетании с некоторыми знаниями «тяжелого программирования» о входящих тегах и классах, которые мы хотим вывести (представьте целую кучу переключателей или выражений `if/then`). Однако мы хотим сделать кое-что получше и создать более общий конструктор модели, чем размещение нашего XML в классы по именам. `SAXModelBuilder`, который мы создали в этой части, получает SAX-события из синтаксически проанализированного XML-файла и динамически создает объекты или наборы свойств в соответствии с именами тегов элемента. Наш конструктор модели небольшой, но он обрабатывает большинство распространенных структур: вложенные элементы и элементы с простым текстовым или численным контентом. Мы относимся к атрибутам как к эквиваленту данных элемента, насколько могут позволить классы нашей модели и поддерживаем три основных типа: `String`, `Double` и `Enum`.

Ниже представлен код:

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.util.*;
import java.lang.reflect.*;
public class SAXModelBuilder extends DefaultHandler
{
Stack<Object> stack = new Stack<>();
public void startElement( String namespace, String localname, String
qname,
Attributes atts ) throws SAXException
{
// Создает новый элемент и устанавливает в него любые атрибуты
Object element;
try {
String className = Character.toUpperCase( qname.charAt( 0 ) ) +
qname.substring( 1 );
element = Class.forName( className ).newInstance();
} catch ( Exception e ) {
element = new StringBuffer();
}
for( int i=0; i<atts.getLength(); i++) {
try {
setProperty( atts.getQName( i ), element, atts.getValue( i ) );
} catch ( Exception e ) { throw new SAXException( "Ошибка: ", e ); }
}
stack.push( element );
}
public void endElement( String namespace, String localname, String qname )
throws SAXException
{
//Добавляет элемент к родительскому
if ( stack.size() > 1) {
Object element = stack.pop();
try {
setProperty( qname, stack.peek(), element );
} catch ( Exception e ) { throw new SAXException( "Ошибка: ", e ); }
}
}
public void characters(char[] ch, int start, int len )
{
```

```
//Возвращает текстовый контекст элемента
String text = new String( ch, start, len );
if ( text.trim().length() == 0 ) { return; }
((StringBuffer)stack.peek()).append( text );
}
void setProperty( String name, Object target, Object value )
throws SAXException, IllegalAccessException, NoSuchFieldException
{
Field field = target.getClass().getField( name );
// Преобразует значение в тип поля
if ( value instanceof StringBuffer ) {
value = value.toString();
}
if ( field.getType() == Double.class ) {
value = Double.parseDouble( value.toString() );
}
if ( Enum.class.isAssignableFrom( field.getType() ) ) {
value = Enum.valueOf( (Class<Enum>)field.getType(),
value.toString() );
}
// Применяет к полю
if ( field.getType() == value.getClass() ) {
field.set( target, value );
} else
if ( Collection.class.isAssignableFrom( field.getType() ) ) {
Collection collection = (Collection) field.get( target );
collection.add( value );
} else {
throw new RuntimeException( "Невозможно установить свойство..." );
}
}
public Object getModel() { return stack.pop(); }
}
```

На первый взгляд код может показаться немного сложным для восприятия: мы используем рефлексию для создания объектов и установки свойств в поля. Но суть в том, что на самом деле только три метода, `startElement()`, `characters()`, и `endElement()`, вызываются в ответ на входящие теги, и мы сохраняем данные, когда получаем их. Давайте взглянем поближе. Класс `SAXModelBuilder` расширяет класс `DefaultHandler`, помогая нам реализовать интерфейс `ContentHandler`. Поскольку SAX-со-

бытия придерживаются иерархической структуры XML-документа, мы используем простой стек, чтобы отслеживать, какой объект мы в настоящее время синтаксически анализируем.

В начале каждого элемента конструктор модели пытается создать экземпляр класса с тем же именем, что и элемент в верхнем регистре, и толкает его вверх стека.

Каждый вложенный открывающий тег создает новый объект в стеке, пока мы не обнаружим закрывающий тег. До достижения конца элемента мы выталкиваем текущий объект из стека и пробуем применить его значения к его родительскому элементу (окружающий XML-элемент), который становится новой вершиной стека.

Для элементов с простым контекстом, которые не имеют соответствующих классов, мы помещаем `StringBuffer` в стек отдельно для хранения символьного контента, пока тег не закроется. В этом случае имя тега отмечает свойство родителя, который должен получить текст, и, увидев закрывающий тег, мы обрабатываем его тем же способом. Атрибуты применимые к текущему объекту стека методом `startElement()` мы обрабатываем, используя похожую технику. Конечный закрывающий тег вытаскивает элемент верхнего уровня (в данном случае `inventory`) из стека для получения.

Для установки значений в наши объекты мы используем собственный метод `setProperty()`. Он использует рефлексию для поиска поля, совпадающего с именем тега в определенном объекте. Он также обрабатывает несколько простых преобразований типов на основе типа найденного поля.

Если поле типа `Double`, мы анализируем текст, как число; если его тип `Enum`, мы находим соответствующее перечисляемое значение представленное текстом. Наконец, если поле — это не просто поле, а `Collection`, представляющий последовательность XML, тогда мы вызываем его метод `add()` для добавления дочернего элемента в коллекцию вместо попытки присвоения самого поля.

Тест-драйв

Наконец, мы можем испытать наш конструктор модели, используя следующий класс `TestSAXModelBuilder`, который вызывает синтаксический анализатор SAX, устанавливая экземпляр класса `SAXModelBuilder`,

как обработчик контента. Тестовый класс затем выводит некоторую информацию, полученную из файла *zooinventory.xml*:

```
import org.xml.sax.*;
import javax.xml.parsers.*;
public class TestSAXModelBuilder
{
public static void main( String [] args ) throws Exception
{
SAXParserFactory factory = SAXParserFactory
            newInstance();
SAXParser saxParser = factory.newSAXParser();
XMLReader parser = saxParser.getXMLReader();
SAXModelBuilder mb = new SAXModelBuilder();
parser.setContentHandler( mb );
parser.parse( new InputSource("zooinventory.xml") );
Inventory inventory = (Inventory)mb.getModel();
System.out.println("Животные = "+inventory.animal);
Animal cocoa = (Animal)(inventory.animal.get(1));
FoodRecipe recipe = cocoa.foodRecipe;
System.out.println( "Рацион = "+recipe );
}
}
```

вывод должен выглядеть примерно так:

```
Животные = [Сон Фон (млекопитающее, Большая панда), Какао(млекопитающее,
Горилла)]
Рацион = Горилла Чоу: [фрукт, fruit, побеги, листья]
```

В следующей части мы создадим эквивалентный вывод, используя различные инструменты.

Ограничения и возможности

Чтобы сделать наш конструктор модели более завершенным, мы можем использовать более надежное соглашение имен для наших тегов и классов модели (принимая во внимание пакеты и смешанный регистр, и т. д.). Более обобщенно, мы можем ввести произвольные отображения (связи) между именами и классами или свойствами. И, конечно, существует проблема, как использовать нашу модель в обратную сторону, для создания XML-документа. Вы разберетесь далее в этой главе.

XMLEncoder/Decoder

Java включает стандартный инструмент сериализации классов JavaBeans в XML. Классы `XMLEncoder` и `XMLDecoder` пакета `java.beans` аналогичны классам `java.ioObjectInputStream` и `ObjectOutputStream`. Вместо использования родного формата сериализации Java они хранят состояние объектов в формате XML высокого уровня. Мы говорим, что они аналогичны, но XML-кодировщик не полностью замещает сериализацию объекта Java. Напротив, он специализируется на работе с объектами, которые следуют формату шаблона проектирования JavaBeans (устанавливающие сеттеры и получающие геттеры для свойств), и он может только хранить и возвращать состояние объекта, что выражается через открытые свойства bean-компонента.

Когда вы вызываете его, класс `XMLEncoder` пытается создать в памяти копию графа bean-компонентов, которые вы сериализуете, используя только открытые конструкторы и свойства JavaBeans.В процессе работы он пошагово выводит запрашиваемые «инструкции» в XML-формате. Далее `XMLDecoder` выполняет эти инструкции и представляет результат. Основное достижение этого процесса в том, что он крайне устойчив к изменениям реализации класса. Хотя стандартная сериализация Java-объекта может разместить множество видов «совместимых изменений» в классе, ей требуется некоторая помощь разработчика, чтобы сделать это верно.

Поскольку `XMLEncoder` использует только открытые API и пишет простые XML-инструкции, ожидается, что эта форма сериализации будет самым надежным способом для хранения состояния объектов JavaBeans. Процесс называют *долгосрочной устойчивостью* JavaBeans.

Вначале могло показаться, что это устранит необходимость в нашем `SAXModelBuilder` примере.

Почему бы просто не написать наш XML в формате, который `XMLDecoder` понимает, и использовать его для создания нашей модели? Хотя класс `XMLEncoder` очень эффективен в устранении избыточности, как видите, его вывод все еще многословен (в два или три раза длиннее оригинального XML) и не особенно понятен. Хотя можно написать это вручную, но XML-формат создавался не для этого. Наконец, хотя `XMLEncoder` может быть настроен на обработку специальных типов, он страдает от тех же проблем, что и наш конструктор модели, в том, что

«связывание» (пространство имен тегов) определяется строго нашими именами классов Java. Как мы сказали ранее, нам необходим обобщенный инструмент для размещения классов в XML и наоборот.

DOM

В последнем разделе мы использовали SAX для синтаксического анализа XML-документа и создания объектной модели Java. В том примере мы создали особый тип Java для каждого из наших сложных элементов. Если бы мы планировали широко использовать нашу модель в приложении, эта техника давала бы значительную гибкость. Но чаще всего выгоднее (и намного проще) использовать «общую» модель, которая просто представляет содержимое XML в нейтральной форме. Document Object Model (DOM) как раз является такой моделью. DOM API производит синтаксический анализ XML-документа в общем представлении, состоящем из классов, чьи имена такие же, как `Element` и `Attribute`, которые хранят их значения. Вы могли бы использовать это для исследования структуры документа и вытаскивать необходимые вам части способом, который, пожалуй, более удобен, чем низкоуровневый SAX. Недостаток этого способа заключается в том, что анализ и чтение документа происходит в оперативной памяти, хотя для большинства приложений это нормально.

Как мы видели в нашем примере, имея объектную модель, вы можете легко использовать данные. Так что общая DOM кажется привлекательным решением, особенно когда основная работа связана с текстом. Но тут есть одна загвоздка: DOM развивается не так быстро как Java API, и не так хорошо справляется с Java. DOM достаточно совершенен и предоставляет доступ к каждому аспекту оригинального XML-документа, но эта модель общая (и нейтрально-языковая) и потому весьма громоздкая для Java. Позже мы также упомянем JDOM — родную для Java альтернативу DOM, которую более приятно использовать.

DOM API

Классы ядра DOM принадлежат пакету `org.w3c.dom`. Результат синтаксического анализа XML-документа с применением DOM — объект `Document` из этого пакета (см. рис. 24.1). `Document` — это и фабрика и контейнер для иерархического набора объектов `Node`, представляющий структуру документа. Узел имеет родительский узел и может иметь до-

черние узлы, которые могут пересекаться, используя собственные методы `getChildNodes()`, `getFirstChild()`, или `getLastChild()`. Узел также может иметь связанные с ним «атрибуты», которые содержат именную карту узлов.

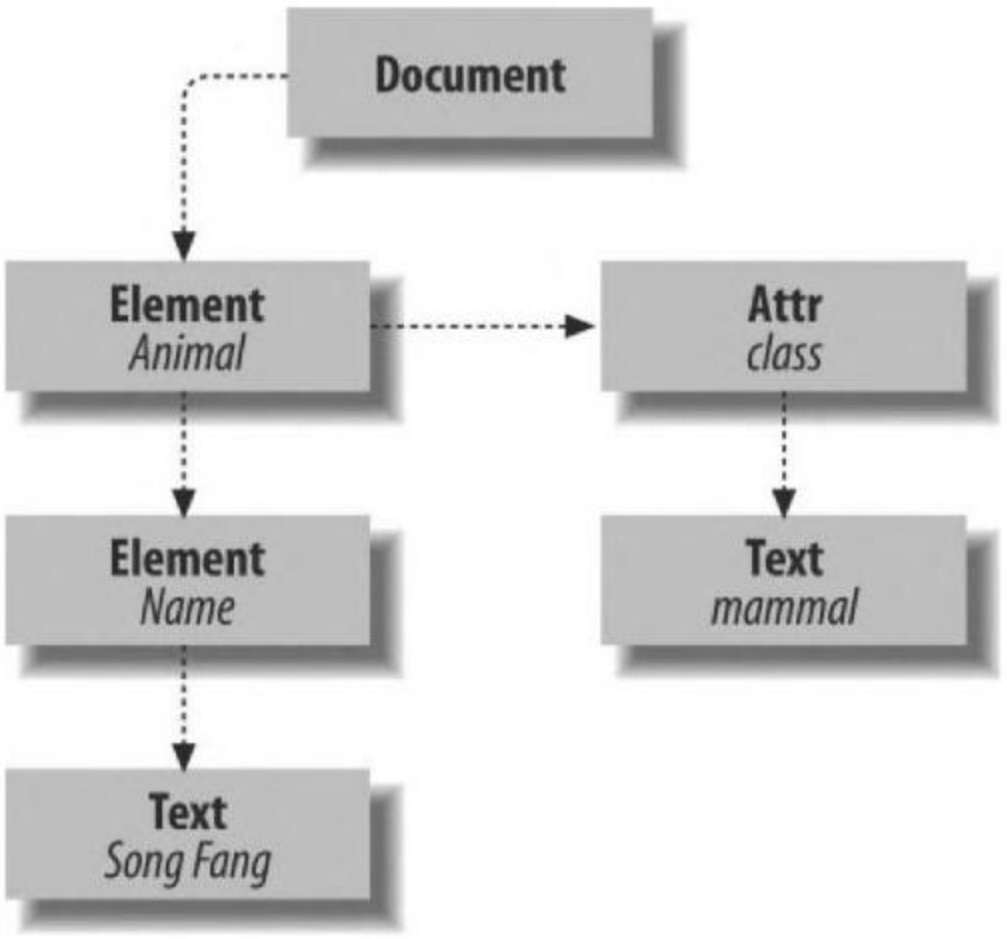

Рис. 24.1. Синтаксический анализ DOM

Подтипы `Node`: `Element`, `Text`, и `Attr`, представляют элементы, текст и атрибуты XML. Некоторые типы узлов (включая эти) имеют текстовые «значения». Например, значение узла `Text` — это текст представленного элемента. То же утверждение верно для `attribute`, `cdata` или `comment` узла. Значение узла можно получить методами `getNodeValue()` и `setNodeValue()`. Мы будем также использовать метод `getTextContent()` объекта `Node`, который извлекает простой текст контекста узла и всех его дочерних узлов.

Подтип `Element` узла предоставляет «случайный» доступ к его дочерним элементам методом `getElementsByTagName()`, который возвращает `NodeList` (простой тип коллекции/набора). Вы можете также получить атрибут по имени из элемента `Element`, используя метод `getAttribute()`.

Пакет `javax.xml.parsers` содержит фабрику для синтаксических анализаторов DOM, так же как и для синтаксических анализаторов SAX. Экземпляр `DocumentBuilderFactory` может быть использован, чтобы создать объект `DocumentBuilder` для синтаксического анализа файла, и выдает в качестве результата объект `Document`.

Тест-драйв DOM

Ниже приведен класс TestDOM:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
public class TestDOM
{
public static void main( String [] args ) throws Exception
{
DocumentBuilderFactory factory = DocumentBuilderFactory
.newInstance();
DocumentBuilder parser = factory.newDocumentBuilder();
Document document = parser.parse( "zooinventory.xml" );
Element inventory = document.getDocumentElement();
NodeList animals = inventory.getElementsByTagName("animal");
System.out.println("Животные = ");
for( int i=0; i<animals.getLength(); i++ ) {
Element item = (Element)animals.item( i );
String name = item.getElementsByTagName( "name" ).item( 0 )
.getTextContent();
String species = item.getElementsByTagName( "species" )
.item( 0 ).getTextContent();
String animalClass = item.getAttribute( "animalClass" );
System.out.println( " "+ name +" ("+animalClass+",
"+species+")" );
}
Element cocoa = (Element)animals.item( 1 );
Element recipe = (Element)cocoa.getElementsByTagName( "foodRecipe" )
.item( 0 );
String recipeName = recipe.getElementsByTagName( "name" ).item( 0 )
.getTextContent();
System.out.println("Рецепт = " + recipeName );
NodeList ingredients = recipe.getElementsByTagName("ingredient");
for(int i=0; i<ingredients.getLength(); i++) {
System.out.println( " " + ingredients.item( i )
.getTextContent() );
}
}
}
```

Класс `TestDOM` создает экземпляр `DocumentBuilder` и использует его для синтаксического анализа файла *zooinventory.xml*. Мы используем метод `DocumentgetDocumentElement()`, чтобы получить корневой элемент документа, с которого мы и начнем. С этого момента мы запрашиваем все дочерние узлы элемента `animal`. Метод `getElementbyTagName()` возвращает объект `NodeList`, который мы далее используем для итераций по нашему творению. Для каждого `animal` мы используем метод `ElementgetElementsByTagName()`, чтобы извлечь имя и дополнительную информацию о дочернем элементе. Каждый из этих запросов потенциально способен вернуть список согласованных элементов, но мы здесь получаем только один, беря первый возвращенный элемент, и запрашиваем его текстовый контекст. Также мы используем метод `getAttribute()` для извлечения атрибута `animalClass` из элемента.

Далее мы применяем метод `getElementsByTagName()` для извлечения элемента `foodRecipe` со второго элемента `animal`. Мы используем его, чтобы получить `NodeList` для всех тегов, соответствующих `ingredient`, и напечатать их как раньше. Вывод должен содержать ту же информацию, что и наш пример на основе SAX. Но, как видите, компромисс не в том, чтобы создать нашу собственную модель классов и что мы должны страдать из-за использования общей модели и создания кода, который значительно сложнее для чтения и менее гибок.

Создание XML с DOM

Все это время мы использовали SAX и DOM API для синтаксического анализа XML. А что с созданием XML? Конечно, это просто — создать тривиальный XML-документ, просто набрав соответствующие строки. Но, если мы планируем создать сложный документ на лету, мы хотели бы получить некоторую помощь со всеми этими кавычками и закрывающими тегами. Мы также хотели бы проверить нашу модель в сравнении с XML DTD или Schema, прежде чем писать ее. Мы можем сделать следующее: создать представление DOM нашего объекта в памяти и затем трансформировать его в текст. Кроме того это полезно, если мы хотим считать документ и затем внести в него какие-то изменения. Для этого мы используем пакет `java.xml.transform`. Этот пакет делает намного больше, чем просто вывод XML. Как следует из его имени, это часть общих возможностей трансформаций (преобразования). Он подключает языки XSL/XSLT для создания одного XML-документа из другого. (Мы будем говорить об XSL позже в этой главе.) Сейчас мы

не станем обсуждать детали размещения DOM в памяти, но это вполне естественно следует из того, что вы узнали об обходе дерева в нашем предыдущем примере. Следующий пример, PrintDOM, просто выполняет синтаксический анализ нашего файла *zooinventory.xml* в DOM и затем выводит полученный DOM обратно на экран. Такой же код вывода должен выводить любой DOM, считанный с файла или созданный в памяти с использованием DOM-методов `Document` и `Element`, и т. д.

```
import javax.xml.parsers.*;
import org.xml.sax.InputSource;
import org.w3c.dom.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
public class PrintDOM {
public static void main( String [] args ) throws Exception
{
DocumentBuilder parser = DocumentBuilderFactory.newInstance()
.newDocumentBuilder();
Document document = parser.parse(
new InputSource("zooinventory.xml") );
Transformer transformer = TransformerFactory.newInstance()
.newTransformer();
Source source = new DOMSource( document );
Result output = new StreamResult( System.out );
transformer.transform( source, output );
}
}
```

Заметим, что импорт выполняется почти так же долго, как и вся программа! Здесь мы используем экземпляр объекта `Transformer` и его простейшую способность копировать из исходника в вывод. Мы вернемся к объекту `Transformer` позже, когда будем обсуждать XSL, там он будет выполнять много работы за нас.

JDOM

Как мы обещали ранее, сейчас мы опишем более простой DOM API: JDOM, созданный Джейсоном Хантером и Бреттом Маклафлином, двумя авторами из издательства O'Reilly (книги «Java Servlet Programming» и «Java and XML», соответственно). Это более естественный Java DOM,

который использует действительный набор типов Java, таких как `List`, для их иерархии и предоставляет незначительно более удобные методы для создания документов. Вы можете получить последнюю версию JDOM по адресу **www.jdom.org**.

Далее приведена версия JDOM нашей стандартной «тестовой» программы:

```
import org.jdom.*;
import org.jdom.input.*;
import org.jdom.output.*;
import java.util.*;
public class TestJDOM {
public static void main( String[] args ) throws Exception {
Document doc = new SAXBuilder().build("zooinventory.xml");
List animals = doc.getRootElement().getChildren("Animal");
System.out.println("Животные = ");
for( int i=0; i<animals.size(); i++ ) {
String name = ((Element)animals.get(i)).getChildText("Name");
String species = ((Element)animals.get(i))
.getChildText("Species");
System.out.println( " "+ name +" ("+species+")" );
}
Element foodRecipe = ((Element)animals.get(1))
.getChild("FoodRecipe");
String name = foodRecipe.getChildText("Name");
System.out.println("Рацион = " + name );
List ingredients = foodRecipe.getChildren("Ingredient");
for(int i=0; i<ingredients.size(); i++)
System.out.println( " "+((Element)ingredients.get(i))
.getText() );
}
}
```

Класс `Element` JDOM обладает некоторыми удобными `getChild()` и `getChildren()`, так же как и метод `getChildText()` для извлечения текста узлов по имени элемента.

Теперь, когда мы изучили основы SAX и DOM, мы перейдем к рассмотрению нового API, который, в определенном смысле, колеблется между этими двумя. `XPath` позволяет нам получать только ту часть документа, которую мы хотим, и предоставляет нам возможность получать эти компоненты в форме DOM.

XPath

XPath — язык выражений для направления частей XML-документа. Вы можете рассматривать XPath-выражения как разновидность регулярных выражений для XML. Они позволяют вам выводить части XML документа на основе шаблонов. В случае с XPath шаблоны больше связаны со структурной информацией, чем с символьным контекстом, и возвращаемые значения могут быть просто текстом или «живым» узлом DOM. С XPath мы можем запросить у XML-документа все элементы с определенным именем или находящиеся в определенных отношениях «родитель-наследник».

Мы можем также применить крайне сложный тест или *предикат* к узлам, которые позволят нам создать сложный запрос, такой как этот: дай мне все элементы `Animal` с атрибутами `Weight` свыше 400, `Temperament` со значением `irritable`, и `AnimalClass` — со значением `mammal`.

В полную спецификацию XPath входит много функций, она также включает компактный и более многословный синтаксис. Мы не будем пытаться охватить все это, но основы знать просто необходимо потому, что XPath выражения присутствуют в ядре XSL преобразований и других API, которые относятся к частям XML-документов. Полная спецификация — не лучшее чтение перед сном, но вы можете найти ее здесь **www.w3.org/TR/xpath**.

Узлы

Выражения XPath обращаются к `Node` в дереве XML-документа. Узел может быть элементом (возможно, имеющим дочерние узлы), как `<animal>...</animal>` или узлом низкого уровня документа представленным атрибутами (например, `animalClass="mammal"`), блоком CDATA, или даже комментарием. Используя синтаксис XPath можно получить доступ ко всей структуре XML-документа. Как только мы обратились к узлу, мы можем или привести контекст в текстовую строку (как мы могли бы сделать с простым текстовым контекстом, таким как name) или обратиться к нему как к собственному дереву DOM для дальнейшего чтения или манипуляций с ним.

В табл. 24.2 приведен самый основной синтаксис, связанный с узлами. К узлам обращаются по имени пути со слеш-разделителями. Например, `/Inventory/Animal` связан с набором всех узлов `Animal` внутри узла

Inventory. Если нам нужен список имен всех узлов Animal, мы должны использовать /Inventory/Animal/Name. Синтаксис // отмечает узел где-либо в документе, на любом уровне расположения, так //Name будет указывать имена элементов Animal, FoodRecipes, и, возможно, многих других элементов. Для большей точности мы могли использовать //Animal/Name, чтобы выбрать только элементы Name, родителем которых является элемент Animal. Символ @ сопоставляет атрибуты. Это становится более полезным при использовании предикатов, которые мы опишем далее. Наконец, знакомые обозначения . и .. могут быть использованы для «перемещения» по узлу, продолжайте чтение, чтобы понять как их использовать.

Табл. 24.2. Основной синтаксис, связанный с узлами

Синтаксис	Пример	Описание
/Name	/inventory/animal	Все узлы animal внутри /inventory.
//Name	//animal	Все узлы animal где-либо в документе. foodRecipe/animal также будет включен.
Name/*	/inventory/*	Все дочерние узлы, узла inventory (animal и другие элементы расположенные прямо под inventory).
@Name	//animal/@animalClass	Все атрибуты animalClass узлов animal.
.	/inventory/animal/.	Текущий узел (все узлы animal).
..	/inventory/animal/..	Родительский узел (inventory).

Предикаты

Предикаты могут позволить нам применять проверку к узлу. Узлы, которые проходят проверку, включаются в итоговый набор или используются для выбора других узлов (дочерних или родительских), связанных с ними. В XPath множество типов тестов. В табл. 24.3 приведены некоторые примеры.

Предикаты могут быть скомпонованы (AND) с использованием этого синтаксиса или просто добавлением большего количества предикатов, как здесь:

```
//animal[@animalClass="млекопитающее"][weight > 400]
```

Здесь мы запрашиваем узлы animals с атрибутом класса «млекопитающее» и элементом weight, содержащим число большее 400.

Табл. 24.3. Предикаты

Синтаксис	Пример	Описание
`[n]`	`/inventory/animal[1]`	Выбирает *n-ый* элемент из множества (начинается с 1, не с 0). Например, выбрать первый элемент `animal` в узле `inventory`.
`[@name=value]`	`//animal[@animalClass= "млекопитающее"]`	Выбирает узлы с указанным значением атрибута. Например, узлы `animal`, у которых атрибут `animalClass` имеет значение «млекопитающее».
`[element=value]`	`//animal[name="Какао"]`	Отмечает узлы с узлами потомками, чье текстовое значение совпадает с указанным. Например, отмечает узлы `animal` с элементами `name` содержащими простой текст «Какао».
`=!=>`	`< //animal[weight > 400]`	Предикаты также могут выполнять проверку на неравенство или численное сравнение с указанным значением.
`and, or`	`//animal[@animalClass= "млекопитающее" or @ class="рептилия"]]`	Предикаты могут использовать логические операции `AND` и `OR` для проверки. Например, узлы `animal`, чей атрибут `animalClass` или млекопитающее, или рептилия.

Теперь мы можем также увидеть пользу оператора . . . Предположим, мы хотим найти все узлы `animal` с атрибутом `foodRecipe`, который использует фрукт как `ingredient`:

```
//animal/foodRecipe[ingredient="фрукт"]/..
```

. . значит, что вместо извлечения самого узла `foodRecipe`, мы возвращаем его предка — элемент `animal`. Оператор . (текущий узел) полезен в других случаях, когда мы используем функции `Xpath`, чтобы управлять значениями более изящно. Мы скажем несколько слов о функциях далее.

Функции

Спецификация XPath включает в себя не только обход основных узлов и синтаксис предикатов, который мы показали, но также возможность вовлекать более открытое множество функций, работающих с узлами и их контекстом. Эти функции XPath охватывают широкий спектр обязанностей, и здесь мы покажем лишь пару примеров. Функции делятся на несколько общих категорий. Некоторые функции выбирают типы узлов, отличные от элементов. Например, не применяя спе-

циального синтаксиса для выбора комментариев в XML, вы вызываете специальный метод `comment()` следующим образом:

```
/inventory/comment()
```

Это выражение возвращает любые узлы XML-комментариев, которые являются потомками элемента `inventory`. Кроме того XPath предоставляет функции, дублирующие весь (компактный) синтаксис, который мы обсуждали, добавляя такие методы, как `child()` и `parent()` (соответственно `.` и `..`). Другие функции обращающиеся к узлам, например `last()` и `count()`.

```
/inventory/animal[last()]
```

Это выражение выбирает последний элемент-потомок элемента `animal` в `inventory`, схожим образом с предикатом `[n]`, выбирающим *n-ый* элемент.

```
//foodRecipe[count(ingredient)>2]
```

Это выражение выбирает все элементы `foodRecipe`, содержащие более двух ингредиентов (круто, не так ли?).

Наконец, существует много строковых функций. Некоторые полезны для простой проверки, в то время как другие действительно полезны только в контексте XSL и бесполезны вне языка с базовым форматированием и управлением строками. Например, методы `contains()` и `starts-with()` могут быть использованы для поиска текстовых значений внутри XML-документов:

```
//animal[starts-with(name,"C")]
```

Это выражение выбирает элементы `animals`, чей атрибут `name` начинается с символа `C` (например, Сон Фон). Метод `contains()` может быть использован похожим образом для поиска подстроки в тексте.

XPath API

Теперь, когда мы попробовали на вкус синтаксис, давайте рассмотрим, как использовать API. Процедура похожа на регулярные выражения Java API для строк. Мы используем фабрику для создания `XPath` объекта. Затем мы можем или вычислить выражение, или «свести» выражение к `XPathExpression` для лучшего представления, если мы будем использовать его более одного раза:

```
XPath xpath = XPathFactory.newInstance().newXPath();
InputSource source = new InputSource( filename );
String result = xpath.evaluate( "//animal/name", source );
// Сон Фон
```

Здесь мы использовали простейшую форму метода `evaluate()`, которая возвращает только первое вхождение и получает значение как строку. Этот метод полезен для извлечения простых текстовых значений из элементов. Однако, если нам необходим полный набор значений (например, имена всех `animal` отмеченных этим условием), нам необходимо вернуть результат как набор `Node` вместо объектов

Возвращаемый тип метода (в переопределенной форме) `evaluate()` контролируется идентификаторами класса `XPathConstants`. Мы можем получить результат в виде: `STRING`, `BOOLEAN`, `NUMBER`, `NODE`, или `NODESET`. По умолчанию это тип `STRING`, который выводит теги элементов потомков и возвращает только текст отмеченных узлов. Типы `BOOLEAN` и `NUMBER` предназначены для обработки примитивных типов. Типы `NODE` и `NODESET` возвращают объекты `org.w3c.dom.Node` и `NodeList`, соответственно. Нам необходим `NodeList` для получения всех значений.

```
NodeList elements = (NodeList)xpath.evaluate(
expression, inputSource, XPathConstants.NODESET );
```

Далее давайте соберем все это вместе в простом полезном примере.

XMLGrep

Этот простой пример может использоваться как утилита командной строки, подобная `grep`, для тестирования XPath выражений с файлами. Она применяет выражение XPath и затем выводит результирующие элементы как XML-текст, используя технологию, схожую с той, которую мы использовали в нашем примере PrintDOM. Узлы, которые не являются элементами (например, атрибуты, комментарии и т. д.), просто выводятся методом `toString()`, которого обычному серверу будет достаточно для их идентификации, но вы можете дополнить пример на свой вкус. А вот и он:

```
import org.w3c.dom.*;
import org.xml.sax.InputSource;
import javax.xml.xpath.*;
import javax.xml.transform.*;
```

```
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
public class XMLGrep {
public static void printXML( Element element )
throws TransformerException {
Transformer transformer =
TransformerFactory.newInstance().newTransformer();
transformer.setOutputProperty( OutputKeys.OMIT_XML_DECLARATION,"yes" );
Source source = new DOMSource( element );
Result output = new StreamResult( System.out );
transformer.transform( source, output );
System.out.println();
}
public static void main( String [] args ) throws Exception {
if ( args.length != 2 ) {
System.out.println( "Обращение: PrintXPath выражение file.xml" );
System.exit(1);
}
String expression = args[0], filename = args[1];
XPath xpath = XPathFactory.newInstance().newXPath();
InputSource inputSource = new InputSource( filename );
NodeList elements = (NodeList)xpath.evaluate(
expression, inputSource, XPathConstants.NODESET );
for( int i=0; i<elements.getLength(); i++ )
if ( elements.item(i) instanceof Element ) {
printXML( (Element)elements.item(i) );
} else
System.out.println( elements.item(i) );
}
}
```

И вновь в этом примере мы используем много импорта. Преобразование кода в методе `printXML()` основано на примере PrintDOM с единственным дополнением. У нас есть набор свойств преобразователя для отсечения стандартного определения XML, которое обычно выводится в заголовке документа. Поскольку мы можем выводить более одного (корневого) элемента, вывод не в полной мере сформирован как XML.

Запустите пример, передавая XPath выражение и имя XML-файла в качестве аргументов:

```
% java XMLGrep "//animal[starts-with(name,'C')]" zooinventory.xml
```

Этот пример действительно полезен для того, чтобы испытать XPath. Пожалуйста, попробуйте его несколько раз. Освоение этих выражений (и дальнейшее изучение) даст вам большую власть над XML-документами и, опять же, базовые знания о преобразованиях XSL.

XInclude

XInclude — очень простое средство «импорта» XML-документов. Используя XInclude-директиву, вы можете просто включить один XML-документ в другой как XML или как простой (или управляющий) текст. Это означает, что вы можете разделить ваши документы на удобное количество файлов. Мы должны отметить, что это можно сделать и иным образом, используя полную декларацию XML, но это приводит к проблемам. XInclude проще и делает то, что следует из его названия, — подключает указанный документ в текущее размещение; вам необходимо лишь определить нужное пространство имен для нового элемента `<include>`. Пример:

```
<Book xmlns:xi="http://www.w3.org/2001/XInclude">
<Title>Learning Java</Title>
<xi:include href="chapter1.xml"/>
<xi:include href="chapter2.xml"/>
<xi:include href="chapter3.xml"/>
...
</Book>
```

Мы используем идентификатор пространства имен `xi` для квалификации элементов `<include>`, которые мы применяем для импорта глав нашей книги. По умолчанию файл импортирован в виде XML-контекста — это означает, что синтаксический анализатор встроен в добавляемый документ как часть нашего документа. Результаты представлений DOM или SAX будут отображать соединяемые документы как единое целое. С другой стороны, мы можем использовать атрибут `parse` для определения того, что мы нацелены добавить только текст. В этом случае текст автоматически скрывается от нас как секция CDATA. Например, мы можем использовать это, чтобы включить пример XML в нашу книгу, не опасаясь, что он будет интерпретирован как часть файла:

```
<Example>
<Title>Пример инвентаризации в зоопарке</Title>
<xi:include parse="text" href="zooinventory.xml"/>
</Example>
```

Здесь весь *zooinventory.xml* файл будет подключен как аккуратно изолированный от нас текст (не добавленный к нашему документу как XML). XInclude также предоставляет «замещающий» контекст с использованием вложенного элемента `fallback`. Элемент `fallback` может указывать на другой файл или просто содержать XML на случай, если подключаемый файл невозможно найти. Например:

```
<xi:include parse="text " href="zooinventory.xml">
<xi:fallback href="filenotfound.xml"/>
</xi:include>
<xi:include parse="text" href="example.xml">
<xi:fallback>Этот пример потерян...</xi:fallback>
</xi:include>
```

В первом случае, если файл *zooinventory.xml* не найден, будет подключен файл *filenotfound.xml*. Во втором случае, «упущенный» текст будет добавлен вместо файла. Если не указан элемент `fallback`, произойдет критическая ошибка синтаксического анализа. Пустой элемент `fallback` может использоваться для предупреждения любых ошибок. Элементы `fallback` могут быть также объединены для комбинирования их поведения.

Подключение XInclude

Перед тем как мы начнем синтаксический анализ нашего файла, нам надо настроить пару переключателей для включения автоматической обработки с XInclude. Во-первых, так как XInclude устанавливает пространство имен пользователя, мы должны настроить обработку пространства имен в нашем механизме синтаксического анализа. Во-вторых, мы должны явно сообщить программе синтаксического анализа, как интерпретировать директивы подключения. Для изменения нашего примера `PrintDOM` таким образом, чтобы перед выводом результата представлялись включения, мы установим эти переключатели фабрики перед созданием экземпляра `DocumentBuilder`:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
// включение обработки XInclude
factory.setNamespaceAware( true );
factory.setXIncludeAware( true );
DocumentBuilder parser = factory.newDocumentBuilder();
Document document = parser.parse( input );
```

Обе функции на сегодняшний день должны использоваться по умолчанию. Но исторически они появились позже XML и поэтому обрабатываются как специальное дополнение, которое должно быть подключено. Прежде чем двинуться дальше, мы должны заметить, что XInclude допускает использование выражений XPath (с API под названием XPointer), чтобы подключать только определенные части XML-документа.

Проверка документов

Слова, слова, слова — а сердца нет!

Уильям Шекспир, *Троил и Крессида*

В этой части мы расскажем о DTD и XML Schema, двух способах создания правил в XML-документе. DTD — это простой грамматический справочник для XML-документа, определяющий где, в каком порядке, с какими атрибутами и т. д. теги могут появляться. XML Schema — следующее поколение DTD, используя XML Schema, вы можете описывать обрабатываемый контент документа также просто как структуру. XML Schema написана в терминах примитивов: чисел, дат и простых регулярных выражений, а также позволяет пользователю определять сложные типы в стиле грамматики. Слово *schema* означает план или простоту структуры, так что мы можем использовать DTD и XML Schema совместно, как схему, где обе модели применимы.

Хотя возможности DTD намного более ограничены, эта схема все еще широко используется. Возможно, это связано со сложностью ручного применения XML Schema. Стандарт W3C XML Schema многословный и громоздкий, что может объяснить появление нескольких альтернативных синтаксисов. `javax.xml.validation` API представляет XML-верификацию путем применения внешних модулей. Из сред поддерживается только W3C XML Schema, но новые языки схем могут быть добавлены в будущем. Верификация с применением DTD, как более ранняя возможность, поддерживается непосредственно в синтаксическом анализаторе SAX. В этой части мы будем использовать оба варианта.

Использование средств проверки документов

Верификация документов в XML — это ключевое качество, делающее его полезным форматом хранения данных. Использование схемы подобно способу, которым Java-классы соблюдают проверку типов язы-

ка. Схема определяет типы документа. Документы, удовлетворяющие данной схеме, часто называются *документами экземпляра* схемы.

Этот тип безопасности предоставляет уровень защиты, который исключает появления сложных для проверки ошибок кода. Однако верификация не всегда необходима в каждой разработке. Например, когда похожие инструменты создают XML и считывают его вновь в коротком промежутке времени, верификация может не требоваться. Тем не менее это имеет неоценимое значение в процессе разработки. Иногда верификация документа используется во время разработки и отключается в процессе эксплуатации.

Элементы языка DTD

Язык DTD крайне прост. В основном DTD — это набор специальных тегов, которые определяют каждый элемент в документе и для сложных типов предоставляют список элементов, которые они могут содержать. DTD тег `<!ELEMENT>` состоит из имени тега и либо ключевого слова, указывающего на тип данных, либо параметризованного списка элементов.

```
<!ELEMENT Name ( #PCDATA )>
<!ELEMENT Document ( Head, Body )>
```

Специальный идентификатор `#PCDATA` (синтаксически анализируемые символьные данные) указывает строку. При предоставлении списка ожидается появление элементов в этом порядке. Список может иметь вложенный список, и пункты можно создавать, используя вертикальную черту (`|`) как оператор `OR`. Специальное обозначение может также использоваться для указания количества пунктов, которое может появиться, два примера такого обозначения приведены в табл. 24.4.

Табл. 24.4. DTD обозначение определяющее вхождения

Символ	Значение
*	Ноль или более вхождений
?	Ноль или одно вхождение

Атрибуты элементов определяются тегом `<!ATTLIST>`. Этот тег задействует DTD для усиления правил атрибутов. Он принимает список идентификаторов и предустановленных значений: `<!ATTLIST Animal animalClass (неизвестно | млекопитающее | рептилия) " неизвестно">`

Данный тег ATTLIST сообщает, что элемент animal имеет атрибут animalClass, который может принимать одно или несколько значений: неизвестно, млекопитающее, рептилия. По умолчанию это неизвестно. Мы не можем охватить в этом разделе все возможности DTD. Но следующий пример *zooinventory.xml* гарантированно будет соответствовать описанному нами формату. Пометьте следующий файл с названием *zooinventory.dtd* (или скачайте его по ссылке **eksmo.ru/smv/Primers.zip**):

```
<!ELEMENT inventory ( animal* )>
<!ELEMENT animal ( name, species, habitat, (food | foodRecipe),
temperament, weight )>
<!ATTLIST animal animalClass (неизвестно | млекопитающее | рептилия |
птица | рыба ) "неизвестно">
<!ELEMENT name ( #PCDATA )>
<!ELEMENT species ( #PCDATA )>
<!ELEMENT habitat ( #PCDATA )>
<!ELEMENT food ( #PCDATA )>
<!ELEMENT weight ( #PCDATA )>
<!ELEMENT foodRecipe ( name, ingredient+ )>
<!ELEMENT ingredient ( #PCDATA )>
<!ELEMENT temperament ( #PCDATA )>
```

DTD сообщает, что элемент inventory состоит из произвольного количества элементов animal. Элемент animal содержит теги name, species, habitat и затем либо food, либо foodRecipe. Структура foodRecipe дополнительно определена позже.Чтобы использовать DTD, мы ассоциируем его с XML-документом. Сделать это мы можем, разместив определение DOCTYPE в XML-документе, что позволит синтаксическому анализатору XML распознать и обработать его. Верификация Java API, о роли которой в синтаксическом анализе и верификации мы расскажем отдельно в следующей части, может быть использована для верификации произвольного XML вместо любого вида схемы, включая DTD. Проблема в том, что вне среды верификация API реализована только в синтаксисе более новых схем XML. Так что нам придется положиться на синтаксический анализатор в части верификации.

В этом случае, когда синтаксический анализатор находит DOCTYPE, он пытается загрузить DTD и верифицировать документ. Существует несколько форм, которые может принимать DOCTYPE, но мы будем использовать только одну:

```
<!DOCTYPE Inventory SYSTEM "zooinventory.dtd">
```

Синтаксические анализаторы и SAX, и DOM могут автоматически верифицировать документы при чтении, при условии, что документы содержат декларацию DOCTYPE, однако вы должны явно запросить у фабрики синтаксического анализатора предоставить анализатор, который можно верифицировать. Для этого достаточно установить свойство проверки фабрики структурного анализатора равным true перед тем, как вы запросите экземпляр синтаксического анализатора. Например:

```
...
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating( true );
```

И снова: метод `setValidating()` — это более старший и упрощенный способ подключения верификации документов, которые содержат DTD ссылки, и он привязан к синтаксическому анализатору. Новый пакет верификации, который мы обсудим позже, независим от синтаксического анализатора и более гибок.

Вы не должны использовать метод верификации синтаксическим анализатором в сочетании с новым API верификации, за исключением ситуации, когда вы хотите по какой-то причине верифицировать документ дважды.

Попробуйте добавить строку `setValidating()` в наш пример конструктора модели после создания фабрики. Попробуем добавить или удалить из файла *zooinventory.xml* элемент или атрибут, и посмотрим, что произойдет, когда вы запустите пример. Вы должны получить содержательное сообщение об ошибке от синтаксического анализатора, которое указывает проблемы и приводит к ошибке структурного анализа. Для получения большей информации о верификации вы можете зарегистрировать объект `org.xml.sax.ErrorHandler`, используя синтаксический анализатор, но Java по умолчанию устанавливает тот, который автоматически сообщает об ошибке.

XML Schema

Хотя DTD могут определить основу структуры XML-документа, они не могут предоставить достаточно богатый словарь для описания отношений между элементами и сообщить немного больше об их контексте. Например, неразумно использовать DTD, чтобы определить, содержит этот элемент численные или перечисляемые данные или даже чтобы управлять длиной строки. Стандарт XML Schema адресует структуру

и данные контекста XML документа. Это следующий логический шаг и XML Schema (или один из конкурирующих языков схем с похожими возможностями) должен заменить DTD в будущем.

XML Schema приносит эквивалент строгой типизации XML, основываясь на множестве предопределенных типов элементов, и позволяет пользователям определять новые сложные типы на их усмотрение. Эти схемы даже позволяют расширять типы и использовать полиморфизм, как в случае с типами в языке Java. Хотя мы не можем описать все подробности XML Schema, мы приведем эквивалент W3C XML Schema нашего файла *zooinventory.xml* :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="inventory">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="animal"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="name" type="xs:string"/>
<xs:element name="animal">
<xs:complexType>
<xs:sequence>
<xs:element ref="name"/>
<xs:element name="species" type="xs:string"/>
<xs:element name="habitat" type="xs:string"/>
<xs:choice>
<xs:element name="food" type="xs:string"/>
<xs:element ref="foodRecipe"/>
</xs:choice>
<xs:element name="temperament" type="xs:string"/>
<xs:element name="weight" type="xs:double"/>
</xs:sequence>
<xs:attribute name="animalClass" default="неизвестно">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="неизвестно"/>
<xs:enumeration value="млекопитающее"/>
<xs:enumeration value="рептилия"/>
<xs:enumeration value="птица"/>
```

```
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="foodRecipe">
<xs:complexType>
<xs:sequence>
<xs:element ref="name"/>
<xs:element maxOccurs="unbounded" name="ingredient" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Обычно схема может находиться в файле XML Schema Definition, с расширением *.xsd*. Первое, что необходимо отметить: этот файл схемы — обычный, синтаксически верный XML-файл, который использует элементы из пространства имен W3C XML Schema. Здесь мы применяем вложенное описание для определения элементов, которые будут появляться в нашем документе. Как в большинстве языков, существует более одного способа для выполнения этой задачи. Здесь мы избегаем «сложности» элементов `animal` и `foodRecipe` в их собственном отдельном описании элемента и обращаемся к ним в их родительском элементе, используя атрибут `ref`. В данном случае мы сделали это в основном ради удобства чтения; он должен иметь одно большое глубоко вложенное описание элемента, начиная с элемента `inventory`. Однако обращаясь к элементу таким способом, мы также получаем возможность повторно использовать то же описание элемента в нескольких местах в документе. Примером этого является наш элемент `name`. Хотя он делает для нас не так уж много, мы исключаем элемент `name` и обращаемся к нему и через `Animal/Name`, и через `FoodRecipe/Name`. Подобное исключение `name` позволит нам использовать расширенные возможности схемы или описывать правила для `name` в одном месте (например, как долго, какой тип символов разрешен) и повторно использовать, как «тип» где необходимо.

Управление директивами `sequence` и `choice` позволяет нам определить структуру дочерних элементов, а атрибуты `minOccurs` и `maxOccurs` — мощность (как много экземпляров). Директива `sequence` сообщает, что близкие элементы должны появляться в определенном порядке (если они затребованы). Директива `choice` позволяет нам определять альтернативные дочерние элементы, как `food` или `foodRecipe`. Мы определяем

допустимые значения для нашего атрибута `animalClass`, используя объявление `restriction` и теги `enumeration`.

Простые типы

Хотя мы на самом деле не испытывали его здесь, атрибут `type` наших элементов затрагивает стандартизацию типов XML Schema. Все наши «текстовые» элементы определены типом `xs:string`, стандартом строкового типа XML Schema (вид эквивалента PCDATA в нашем DTD). Существует множество других стандартных типов, охватывающих такие вещи, как даты, время, периоды, числа и даже URL. Они называются *простыми типами* (хотя некоторые из них не такие уж и простые) исходя из того, что они стандартизированные или «встроенные». В табл. 24.5 приведен список простых типов W3C Schema и соответствующих им типов Java. Перечисленное станет интереснее позже, когда мы расскажем о JAXB и автоматическом связывании XML с Java классами.

Табл. 24.5. Простые типы W3C Schema

Тип элемента Schema	Тип Java	Пример
xsd:string	java.lang.String	"Это текст"
xsd:Boolean	boolean	true, false, 1, 0
xsd:byte	byte	
xsd:unsignedByte	short	
xsd:integer	java.math.BigInteger	
xsd:int	int	
xsd:unsignedInt	long	
xsd.long	long	
xsd:short	short	
xsd:unsignedShort	int	
xsd:decimal	java.math.BigDecimal	
xsd:float	float	
xsd:double	double	
xsd:Qname	javax.xml.namespace.QName	funeral:corpse
xsd:dateTime	java.util.Calendar	2004-12-27T15:39:05.000-06:00
xsd:base64Binary	byte[]	PGZv
xsd:hexBinary	byte[]	FFFF
xsd:time	java.util.Calendar	15:39:05.000-06:00
xsd:date	java.util.Calendar	2004-12-27
xsd:anySimpleType	java.lang.String	

Например, у нас есть элемент `weight` со значением с плавающей точкой, как этот в нашем элементе `animal`:

```
<Weight>400.5</Weight>
```

Сейчас мы можем верифицировать его в нашей схеме, добавив следующую запись в соответствующее место:

```
<xs:element name="weight" type="xs:double"/>
```

XML Schema не только следит за тем, чтобы содержание элементов соответствовало этим простым типам, но и может дать нам больше контроля над текстом и значениями элементов в нашем документе, с помощью простых правил и шаблонов, аналогичных регулярным выражениям.

Сложные типы

Дополнительно к предопределенным простым типам, перечисленным в табл 24.5, мы можем определить собственные *сложные типы* в нашей схеме. Сложные типы — это типы элементов, которые имеют внутреннюю структуру и возможно дочерние элементы. Наши элементы `inventory`, `animal`, и `foodRecipe` сложного типа и их контекст должен быть объявлен тегом `complexType` в нашей схеме. Определение сложного типа может быть использовано повторно, подобно определениям элементов в нашей схеме; то есть мы можем вырвать определение сложного типа и дать ему имя. Мы можем затем обратиться к типу по имени в атрибуте `type` других элементов. Поскольку каждый из наших сложных типов будет использоваться однажды в соответствующем ему элементе, мы не даем ему имени. Определение и использование в одном месте, соответствуют *анонимному определению типов*. Например, мы могли бы отделить наш тип `animal` от определения его элемента, как здесь:

```
<xs:element name="inventory">
<xs:complexType>
<xs:sequence>
<xs:element name="animal" maxOccurs="unbounded" type="AnimalType"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:complexType name="AnimalType">
<xs:sequence>
<xs:element ref="name"/>
```

```
<xs:element name="species" type="xs:string"/>
<xs:element name="habitat" type="xs:string"/>
...
```

Определение `AnimalType` отделено от определения экземпляра элемента `animal`, что позволяет нам использовать другие, иначе названные элементы с той же структурой. Например, наш элемент `inventory` может содержать другой элемент `mainAttraction`, типа `animal`, но с иным именем тега.

Можно намного больше сказать о W3C XML Schema, и он может справиться с более сложными задачами, чем в нашем простом примере. Однако и те фрагменты, которые показали, могут дать вам немало. Вначале вам могут помочь некоторые инструменты. Сейчас мы расскажем об одном из них называемом Trang. Больше информации о XML Schema вы найдете на сайте W3C или в книге «XML Schema» Эрика ван дер Влиста (O'Reilly). В следующей части мы покажем, как верифицировать файл или модель DOM с XML Schema, которую мы создадим, используя новый верификационный API.

Создание схемы из XML-образцов

Многие инструменты могут помочь вам в создании XML Schema. Один из полезных инструментов называется Trang. Это часть проекта альтернативного языка схем, называемого RELAX NG (который мы упомянем позже в этой главе), но Trang очень полезен и сам по себе. Это инструмент с открытым исходным кодом, который может не только осуществлять обмен между DTD и XML Schema, но также создавать необработанный DTD или XML Schema считывая «пример» XML-документа. Это отличный способ наметить базу, основываясь на схеме ваших документов.

Верификация API

Для использования XML-схемы нашего примера, нам необходимо поупражняться с новым API `javax.xml.validation`. Как мы говорили ранее, верификация API — это альтернатива простой верификации, на основе синтаксического анализа поддерживаемого методом `setValidating()` фабрик синтаксических анализаторов. Для использования пакета верификации, мы создаем экземпляр `SchemaFactory`, определяя язык схемы. Далее мы можем провести верификацию DOM или потокового источника со схемой. Следующий пример, Validate представлен в форме простой утилиты командной строки, которую вы можете использовать для

тестирования вашего XML и схем. Просто передайте ей путь к файлу XML и файл схемы XML (*.xsd* файл) как аргументы.

```
import javax.xml.XMLConstants;
import javax.xml.validation.*;
import org.xml.sax.*;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
public class Validate
{
public static void main( String [] args ) throws Exception {
if ( args.length != 2 ) {
System.err.println("Обращение: Верификация xmlfile.xml xsdfile.xsd");
System.exit(1);
}
String xmlfile = args[0], xsdfile = args[1];
SchemaFactory factory =
SchemaFactory.newInstance( XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = factory.newSchema( new StreamSource( xsdfile ) );
Validator validator = schema.newValidator();
ErrorHandler errHandler = new ErrorHandler() {
public void error( SAXParseException e ) {
System.out.println(e);
}
public void fatalError( SAXParseException e ) {
System.out.println(e);
}
public void warning( SAXParseException e ) {
System.out.println(e);
}
};
validator.setErrorHandler( errHandler );
try {
validator.validate( new SAXSource(
new InputSource("zooinventory.xml") ) );
} catch ( SAXException e ) {
// Неверный документ, нет обработки ошибки
}
}
}
```

Поддерживаются типы схем, первоначально перечисленные как константы в классе `XMLConstants`.

Сейчас реализована только W3C XML Schema, и, конечно, в ней есть и другие интригующие типы, которые мы упомянем позже. Наш пример верификации основан на шаблоне, который мы рассматривали ранее, создавая фабрику, и затем экземпляр `Schema`. `Schema` представляет грамматику и создает экземпляр класса `Validator`, который выполняет работу по проверке структуры документа. Здесь мы вызываем метод `validate()` с параметром `SAXSource`, который поступает из нашего файла, но мы также могли использовать `DOMSource` для проверки представления DOM в памяти:

```
validator.validate( new DOMSource(document) );
```

Любые обнаруженные ошибки будут вызывать метод верификации через `SAXException`, но это лишь грубое средство обнаружения ошибок. В целом, и как мы показываем в этом примере, мы хотим зарегистрировать объект `ErrorHandler` с `validator`. Обработчику ошибки может быть сообщено о множестве ошибок в документе и предоставлено больше информации. Когда представлен обработчик ошибок, исключения проходят через него, а не через метод верификации.

Ошибки, создаваемые этим синтаксическим анализатором, могут быть менее критичными. В некоторых случаях ошибки не могут указывать номер строки потому, что верификация необязательна для обработки потока.

Альтернативные языки схем

В дополнение к DTD и W3C XML Schema сегодня используется несколько других популярных языков схем. Одна интересная альтернатива, исключительно ссылающаяся на класс `XMLConstants`, называется RELAX NG. Этот язык схемы предлагает наиболее широко используемые особенности XML Schema в более удобочитаемом формате. Фактически он предоставляет и очень компактный не-XML-синтаксис и регулярный XML-синтаксис. RELAX NG не предоставляет похожие текстовые шаблоны и верификацию значений, как W3C XML Schema. Напротив, эти особенности проверки он оставляет другим инструментам (многие рассматривают это как «бизнес-логику», в любом случае более уместно реализованную вне схемы). Если вы заинтересованы в исследовании других языковых схем, не забудьте испытать RELAX NG и его полезную утилиту преобразования схемы Trang.

Связывание и генерация кода JAXB

Мы сказали, что наша конечная цель в этой главе — автоматическая привязка XML к Java-классам. Сейчас мы обсудим стандартный Java API для XML Binding, JAXB. (Не следует путать с JAXP, API синтаксического анализатора.) JAXB — это стандартное расширение в комплекте с Java 6 и более поздними версиями.

При использовании JAXB разработчику не требуется создавать «хрупкий» код синтаксического анализа. XML-схема или Java-код могут быть использованы, как начальная точка преобразования XML в Java и обратно. (поддерживаются и “Schema first”, и “code first”). Используя JAXB, вы можете отметить ваши Java-классы простыми описаниями, которые размещают (связывают) их с XML, или начать с XML-схемы и создать простые Java-классы (POJO) с включением необходимой аннотации. Вы можете даже вывести XML-схему из ваших Java-классов для использования в качестве начальной точки или контакта с не Java-системами.

При выполнении JAXB может считать XML-документ и выполнить синтаксический анализ в модель, которую вы определили, или вы можете пойти иным путем, наполнив вашу объектную модель и затем записав ее в XML. В обоих случаях JAXB может верифицировать данные, чтобы быть уверенным в их соответствии схеме.

Это может выглядеть как интерфейс DOM, но в этом случае мы не используем обобщенные классы, мы используем собственную модель. В этой части мы снова возьмем классовую модель, созданную для примера SAX из файла *zooinventory.xml*. Мы будем использовать непосредственно знакомые классы `Inventory`, `Animal`, и `FoodRecipe`, но в этот раз, вы увидите, что мы больше сосредоточены на схеме и именах и меньше на автоматическом синтаксическом анализе.

Описание нашей модели

JAXB обеспечивает большую гибкость в размещении наших Java-классов в XML элементах, и существует много особых случаев. Но, если мы принимаем наиболее стандартное поведение нашей модели, мы можем начать с очень небольшой работы. Давайте начнем с того, что разместим классы инвентаризации нашего зоопарка и добавим необходимые аннотации для связи через JAXB их и XML:

```
@XmlRootElement
public class Inventory {
public List<Animal> animal = new ArrayList<>();
}
```

Что ж, это было легко! Да, фактически, как мы намекнули в начале главы, всего лишь добавив описание `@XmlRootElement` к «верхнему уровню» или корневому классу нашей модели, мы получим почти такой же XML, какой использовали ранее. Для создания XML мы применим следующий набор тестов:

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
public class TestJAXBMarshall
{
   public static void main( String [] args ) throws JAXBException {
      Inventory inventory = new Inventory();
      FoodRecipe recipe = new FoodRecipe();
      recipe.name = "Горилла Чоу";
   recipe.ingredient.addAll( Arrays.asList( "листья", "насекомые",
"фрукты" ) );
      Animal animal = new Animal( Animal.AnimalClass.mammal,
"Сон Фон","Большая панда", "Китай", "Бамбук", "Дружелюбный", 45.0, recipe
);
      inventory.animal.add( animal );
      marshall( inventory );
   }
   public static void marshall( Object jaxbObject ) throws JAXBException {
      JAXBContext context = JAXBContext.newInstance(
         jaxbObject.getClass() );
      Marshaller marshaller = context.createMarshaller();
      marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
         Boolean.TRUE);
      marshaller.marshal(jaxbObject, System.out);
   }
}
```

Мы позволили себе добавить несколько конструкторов, чтобы сократить код создания модели, но это не меняет поведение. Это просто четыре строки нашего метода `marshall()`, который на самом деле использует JAXB для вывода в XML. Сначала мы создали `JAXBContext`,

передавая его в тип класса для ранжирования. Мы создали наш метод `marshall()` в какой-то степени многоразовым, получая тип класса из переданных объектов. Однако иногда необходимо передать дополнительные классы методу `newInstance()`, чтобы JAXB были доступны все связные классы, которые могут потребоваться. В этом случае мы просто передали больше типов классов в метод `newInstance()` (он принимает список переменных аргументов, в качестве типов классов). Далее мы создаем объект `Marshaller` из контекста и устанавливаем флаг, который указывает, что мы хотели бы получить хороший, удобочитаемый вывод (по умолчанию вывод идет одной длинной строкой XML). Наконец, мы говорим `marshaller` отправить объект в `System.out`.

Вывод будет выглядеть примерно так:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<inventory>
   <animal>
      <animalClass>млекопитающее</animalClass>
      <name>Сон Фон</name>
      <species>Большая панда</species>
      <habitat>Китай</habitat>
      <food>Бамбук</food>
      <temperament>Дружелюбный</temperament>
      <weight>45.0</weight>
   </animal>
   <animal>
      <animalClass>млекопитающее</animalClass>
      <name>Какао</name>
      <species>Горилла</species>
      <habitat>Центральная Африка</habitat>
      <temperament>Всезнайка</temperament>
      <weight>45.0</weight>
      <foodRecipe>
         <name>Горилла Чоу</name>
         <ingredient>фрукты</ingredient>
         <ingredient>побеги</ingredient>
         <ingredient>листья</ingredient>
      </foodRecipe>
   </animal>
</inventory>
```

Как мы уже сказали, он почти идентичен тому XML, с которым мы работали ранее. Правда, мы решили создать наш XML, используя те же (общие) соглашения, которые использует JAXB, так что это не совсем магия. Первая вещь, которую стоит отметить, это то, что JAXB автоматически переводит имена наших классов в нижний регистр для имен элементов XML (например, класс `Animal` в `<animal>`). Если мы использовали геттеры в стиле JavaBeans вместо общих полей, то же самое было бы верно; например, метод `getSpecies()` будет по умолчанию создавать имя элемента `species`.

Если мы хотели сопоставить имена наших классов и свойств с совершенно другим форматом имен XML, мы могли бы осуществить это, просто используя имя атрибута описаний `@XmlRootElement` и `@XmlElement`. Например, мы могли назвать наш класс `Animal` «creature» (создания) и переименовать `temperament` в «personality» (личность), как здесь:

```
@XmlRootElement(name="creature")
public class Animal
{
...
@XmlElement(name="personality")
public String temperament;
```

Настоящее отличие между нашим созданным XML и нашим предыдущим образцом в том, что атрибут `animalClass` не действует как атрибут. По умолчанию, он сопоставляется с элементом, как и другие свойства класса `Animal`. Мы можем исправить это, используя другое описание: `@XmlAttribute`:

```
public class Animal
{
@XmlAttribute
public AnimalClass animalClass;
...
// Производит...
<inventory>
<animal animalClass="млекопитающее">
<name>Сон Фон</name>
```

Также заметим, что JAXB показывает элемент `food` в первом элементе `animal` и `foodRecipe` — во втором. JAXB будет игнорировать поля или свойства со значением `null` (как в данном случае), если вы не указали, что свойство имеет пустое значение, используя `@XmlElement(nillable=true)`.

Такое поведение автоматически поддерживает чередование между двумя нашими свойствами. Существует много дополнительных описаний, которые предоставляют поддержку сопоставления Java-классов, полей и свойств с другими особенностями XML. В табл. 24.6 мы попытались кратко пояснить, для чего используется каждое описание. Некоторые применения немного сложны, поэтому вы, возможно, захотите обратиться к Javadoc, чтобы уточнить детали.

Табл. 24.6. JAXB-описания

Описание	Определение
`@XmlAccessorOrder`	Используется пакетом или классом, для установки алфавитного порядка или ранжирования полей и свойств. (Порядок по умолчанию не определен.) См. `@XmlType` для самостоятельного определения порядка. Напомним: аннотации уровня пакета в Java размещаются в (отдельном) утверждении пакета в специальном файле, называемом `package-info.java`, в соответствующей структуре пакета.
`@XmlAccessorType`	Используется в пакете или классе, чтобы узнать, ранжированы ли поля и свойства по умолчанию. Вы можете выбрать: только поля, только свойства (получающие/устанавливающие), ничего (только описанное пользователем) или все общие поля и свойства. См. `@XmlTransient` для исключения элементов.
`@XmlAnyAttribute`	Указывает Java объект `Map` для получения любых несвязных XML атрибутных пар имя/значение для сущности (то есть, объект `Map` объединит все оставшиеся атрибуты не соответствующие свойствам и полям, которые найдены).
`@XmlAnyElement`	Указывает Java-объект `List` или `Array` для получения, любых несвязных XML-элементов для сущности (то есть объект `List` объединит все оставшиеся элементы, для которых не найдено соответствующих свойств или полей).
`@XmlAttachmentRef`	Определяет объект `java.activation.DataHandler` для обработки XML MIME-вложений.
`@XmlAttribute`	Связывает Java-поле или свойство с XML-атрибутом. Атрибут `name` может использоваться для определения имени XML-атрибута, отличающегося от имени поля или свойства. Используйте атрибут `required`, чтобы указать требуется ли атрибут.
`@XmlElement`	Связывает Java-поле или свойство с XML-элементом. Атрибут `name` может использоваться для определения имени XML-элемента отличающегося от имени поля или свойства. Используйте атрибут `required`, чтобы указать, требуется ли элемент.
`@XmlElements`	Применяется к коллекции Java, чтобы определить особые имена элемента для содержащихся элементов на основе их Java-типа. Содержит список описаний `@XmlElement` с атрибутами `name` и `type`, которые явно сопоставляют типы Java в коллекции с именами элементов в XML (в нашем примере `inventory` содержит элементы `animal`, поскольку наш список свойств `List`, называется «`animal`»). Если мы выбираем наличие подклассов класса `Animal` в нашей коллекции `inventory`, мы можем сопоставить их с именами XML-элементов `gorilla` и `lemur`. См. `@XmlElementRef`.

Описание	Определение
`@XmlElementRef`	Похоже на `@XmlElements`, используется для создания индивидуальных имен для Java-типов в коллекции. Однако вместо непосредственного определения имен для каждого типа, они определяются при выполнении по типам привязки отдельных Java-типов (в нашем примере, `inventory` содержит элементы `animal`, поскольку наш список `List` называется «`animal`»). Используя `@XmlElementRef`, мы можем определять подклассы `Animal` и наш `inventory` содержит элементы, подобные `gorilla` и `lemur`, с именами, определенными описанием `@XmlRootElement` в соответствующих подклассах. См. важную информацию о связывании классов в `@XmlElementRefs`.
`@XmlElementRefs`	Используется в Java коллекции для предоставления списка описаний `@XmlElementRef`, с атрибутами `type`, явно определяющими Java-типы, которые могут появиться в коллекции. Результат тот же, что при использовании простого описания `@XmlElementRef` для коллекции, но мы явно передаем JAXB имена классов, которые необходимо связать. В другом случае мы должны предоставить полный список входящих классов методу `JAXBContextnewInstance()` для того, чтобы они были распознаны.
`@XmlElementWrapper`	Используется в Java-коллекции для вызова последовательности XML-элементов, чтобы обернуть их в определенный элемент, вместо непосредственного появления прямо в выводе XML (например, наши элементы `animal` появляются прямо в элементе `inventory`). Используя это описание, мы можем вложить их все в новый элемент `animals`.
`@XmlEnum`	Связывает Java `Enum` с XML, позволяет описанию `@XmlEnumValues` использоваться для сопоставления значений типа `enum` в XML, если необходимо.
`@XmlEnumValue`	Связывает отдельное значение Java `Enum` со строкой для использования в XML (например, наше значение `mammal` типа `enum`, может сопоставляться с «`mammalia`»)
`@XmlID`	Поддерживает ссылочную целостность, назначая свойство или поле Java-класса в качестве XML ID атрибута (уникального ключа) для XML-элемента в документе.
`@XmlIDREF`	Поддерживает ссылочную целостность, назначая свойство или поле Java как атрибут `idref`, указывая на элемент с `@XmlID`. Описанное свойство или поле должно содержать экземпляр Java-типа, содержащего описание `@XmlID`. При ранжировании имя атрибута будет именем свойства, и значение будет содержать значение XML ID.
`@XmlInlineBinaryData`	Связывает Java байтовый массив с получением двоичных данных в кодировке base64 из XML.
`@XmlList`	Используется Java-коллекцией для сопоставления элементов с отдельными простыми элементами контекста, посредством разделенного пробелами списка значений вместо серии элементов.
`@XmlMimeType`	Используется Java-типами `Image` или `Source` для определения `MIME` типа в XML в base64 кодировке двоичных данных и привязки к ним.
`@XmlMixed`	Связывает объекты Java-коллекции со «смешанным» XML контентом (например, XML содержащий и текст, и теги элемента в нем) Текст может быть добавлен в коллекцию, как объекты типа `String` разделенные обычными типами Java, представляющими другие элементы.

Описание	Определение
`@XmlRootElement`	Связывает класс Java с XML-элементом, дополнительно предоставляя имя. Это минимальное описание необходимое вашему классу, чтобы сделать возможным размещение его в XML и обратно.
`@XmlElementDecl`	Используется для связи методов XML-схемы с методами объектной фабрики Java, созданной по некоторому сценарию создания кода.
`@XmlRegistry`	Используется с `@XmlElementDecl` при назначении фабрик объектов Java в некоторых сценариях создания кода.
`@XmlSchema`	Связывает Java-пакеты с пространством имен XML по умолчанию.
`@XmlNs`	Используется с `@XmlSchema` для связи Java-пакетов с одним или более XML-префиксов пространства имен.
`@XmlSchemaType`	Применяется к свойству, полю или пакету Java. Определяет Java-тип для использования для стандартных встроенных в XML-схему типов: даты или численный тип.
`@XmlSchemaTypes`	Применяется к Java-пакетам. Хранит список описаний `@XmlSchemaType` сопоставляющих Java-типы встроенными в XML-схему типами.
`@XmlTransient`	Означает, что свойство или поле Java не должно передаваться в XML. Может использоваться в сочетании с передачей всех полей и свойств, по умолчанию, для исключения отдельных элементов. См. `@XmlAccessorType`.
`@XmlType`	Связывает Java-класс с типом схемы XML. Дополнительно атрибут `propOrder` может использоваться, чтобы явно определить порядок, в котором элементы отправляются в XML.
`@XmlValue`	Означает, что Java-свойство или поле содержит «простой» контент XML Java-типа; то есть вместо ранжирования класса как элемента XML, содержащего вложенные элементы для свойства, значение описанного свойства появится прямо в виде контента. Тип Java может иметь только одно свойство, отмеченное как `@XmlValue`.

Создание объекта `Unmarshaller` из XML-кода

Чтобы создать нашу объектную модель из XML, потребуются лишь несколько строк для создания объекта `Unmarshaller` из `JAXBContext` и приведение к нашего корневого элемента к типу Java:

```
JAXBContext context = JAXBContext.newInstance( Inventory.class );
Unmarshaller unmarshaller = context.createUnmarshaller();
Inventory inventory = (Inventory)unmarshaller.unmarshal(
new File("zooinventory.xml") );
```

Класс `Unmarshaller` содержит метод `setValidating()`, как и SAXParser, но применять его не рекомендуется. Вместо него мы могли бы использо-

вать метод `setSchema()` для установки представления XML Schema, если мы хотим верифицировать его как часть процесса синтаксического анализа. Кроме того, мы могли бы просто верифицировать схему отдельно. См. раздел «XML Schema» ранее в этой главе.

Создание модели Java из XML Schema

Если вы начали с XML Schema (файл *xsd*), вы можете создать описанные Java-классы из схемы, используя JAXB инструмент командной строки `xjc`, который поставляется с JDK.

```
xjc zooinventory.xsd
// вывод
Синтаксический анализ схемы...
Компиляция схемы...
создан/Animal.java
создан/FoodRecipe.java
создан/Inventory.java
создан/ObjectFactory.java
```

По умолчанию, вывод размещается в предопределенном пакете в директории с именем *generated*. Вы можете управлять именем пакета, используя переключатель `-p`, и управлять директорией, используя `-d`. Информацию о дополнительных возможностях вы найдете в документации `xjc`. Изучение созданных классов даст вам некоторые подсказки о том, сколько описаний использовано, хотя инструмент `xjc` немного более многословен, чем следовало бы. Также заметим, что `xjc` предоставляет класс, называемый `ObjectFactory`, который содержит фабрики методов для каждого типа, такие как `createInventory()` и `createAnimal()`. Если вы посмотрите на эти методы, вы увидите, что они просто содержат вызов `new` простых Java-объектов и выглядят избыточными. В целом, `ObjectFactory` осталась здесь по традиции. В ранних версиях JAXB, до описаний, создание классов не было таким простым. Кроме того, `ObjectFactory` содержит вспомогательные методы для создания типа `JAXBElement`, который может быть полезен в отдельных ситуациях. По большей части, на это можно не обращать внимания.

Создание XML-схемы из Java-модели

Вы также можете создавать XML Schema прямо из описанных вами Java-классов, используя генератор связей JAXB XML Schema: `schemagen`.

Инструмент командной строки `schemagen` поставляется вместе с JDK. Он может создавать схему начиная с исходных файлов и файлов классов Java.

Используйте аргумент `-classpath`, чтобы определить расположение классов или исходных файлов и затем предоставить имя корневого элемента в вашей иерархии:

```
schemagen -classpath . Inventory
```

Теперь, изучив преобразование XML в Java, мы можем перейти к самостоятельному преобразованию XML с применением XSL, языка стиля для XML.

Преобразование документов с использованием XSL/XSLT

Ранее в этой главе вы использовали объект `Transformer`, чтобы скопировать DOM-представление примера обратно в XML-текст. Мы упоминали, что `Transformer` способен на большее. Сейчас мы расскажем вам все.

Пакет `javax.xml.transform` — это API для применения XSL/XSLT языка преобразования. XSL выступает в роли расширяемого языка стилей таблиц (Extensible Stylesheet Language). Подобно каскадным таблицам стилей (CSS) для HTML, XSL позволяет нам «размечать» XML документы, добавляя теги, которые формируют представление информации. XSL Transformation (XSLT) идет дальше, добавляя возможность полной реструктуризации XML и предоставляя произвольный вывод. XSL и XSLT вместе образуют собственный язык программирования для обработки XML-документа как входящего параметра и получения другого (обычно XML) документа на выходе. С этого момента мы будем называть их обобщенно XSL.

XSL крайне мощен, и новые приложения для его использования появляются каждый день. Например, представьте себе веб-сайт, который часто обновляется и должен быть доступен для различных мобильных устройств и традиционных браузеров. Вместо пересоздания сайта и добавления платформ XSL может преобразовывать контент в соответствующий формат для каждой платформы. Более обобщенно, визуализация контента из XML — просто более удачный способ для представления наших данных и хранения их отдельно от представляемой вами ин-

формации. XSL можно использовать для визуализации всего веб-сайта в разных стилях из файлов, содержащих поток неизменяемых данных XML, например баз данных. В многоязычных сайтах также успешно используется XSL, чтобы текст по-разному отображался для разных аудиторий.

Вы, наверное, уже догадались, о чем мы хотим вас предупредить: XSL — это обширная тема, достойная отдельной книги (см., например, книгу Эрика Бюрка «Java and XSLT»), и мы можем лишь дать вам попробовать на вкус. Кроме того, некоторые находят XSL сложным для понимания с первого раза, так как он требует думать в терминах рекурсивной обработки тегов документа. В последние годы стремление использовать XSL в качестве способа получения веб-контента снизилось из-за распространения применения JavaScript-сценариев на стороне клиента. Однако XSL остается мощным способом трансформации XML и широко используется в других документо-ориентированных приложениях.

Основы XSL

XSL — это стандарт XML, поэтому не должно вызвать удивления, что этот язык основан на XML. *Таблицы стилей XSL* — это XML-документ, использующий специальные теги, определенные пространством имен XSL, для описания преобразования. Большинство основных операций в XSL включают выбор частей входящих XML-документов и создание вывода на основе их контента. Один или несколько *шаблонов* XSL совместно внутри таблицы стилей называются в соответствии с тегами, подающимися на вход. XSL часто используется исключительно для управления вводом, в то время как ввод XML-тегов вызывает вывод в порядке их появления, используя только информацию о их содержании. Но в целом вывод может быть построен из произвольных частей ввода, основываясь на них, как на базе данных, сочетая элементы и атрибуты. XSLT преобразование части XSL добавляет смесь условий и итераций, которые позволяют создавать на основе входных данных любой вывод.

Таблица стилей XSL содержит тег `stylesheet` как собственный корневой элемент. По соглашению таблица стилей определяет префикс пространства имен `xsl` для пространства имен XSL. Внутри таблицы стилей один или более шаблонов тегов содержат атрибут `match`, который описывает элемент, с которым он работает.

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
Я нашел корневой элемент документа!
</xsl:template>
</xsl:stylesheet>
```

Когда шаблон совпадает с элементом, то появляется возможность обработать все его дочерние атрибуты. Простая таблица стилей, показанная здесь, содержит один шаблон, который выбирает корень входящего документа и выводит некий простой текст. По умолчанию, если ввод не выбран, он просто копируется в вывод с раскрытыми тегами (HTML-соглашение). Но здесь мы выбрали корень, тем самым поглотили все входящие данные, и в выводе не появится ничего, кроме нашего сообщения.

Атрибут `match` может обращаться к элементам, используя выражения Xpath, которые мы описали ранее. Это иерархический путь, начинающийся с корневого элемента. Например, `match="/inventory/animal"` выберет только элементы `animal` в файле *zooinventory.xml*. В XSL путь может быть абсолютным и начинаться с «/», или относительным, в этом случае шаблон обнаруживает любое появление тега в любом подконтексте (эквивалент // в XPath). В шаблон мы можем помещать все что угодно, пока остаемся в рамках синтаксически верного XML (в противном случае мы можем использовать секцию `CDATA` или Xinclude). Но самое интересное начинается, когда мы используем части ввода для построения вывода. XSL-тег `value-of` используется для содержимого или дочернего элемента. Например, следующий шаблон должен выбрать элемент `animal` и вывести значение его дочернего элемента `Name`:

```
<xsl:template match="animal">
Имя: <xsl:value-of select="name"/>
</xsl:template>
```

Атрибут `select` использует выражение XPath, относящееся к текущему узлу. В этом случае мы говорим ему вывести значение элемента `name` внутри элемента `animal`. Мы можем использовать относительный путь для более глубоко вложенного внутри `animal` элемента или даже абсолютный путь к части другого документа. Для обращения к «текущему» элементу (в данном случае самому элементу `animal`) выражение `select` может использовать . как путь. Выражение `select` может также извлекать атрибуты из элементов, которые их предоставляют.

Если мы попробуем добавить шаблон `animal` к нашему простому примеру, он не будет ничего выводить.

В чем проблема? Вспомните, мы говорили, что, выбирая элемент, мы имеем возможность обрабатывать все его дочерние элементы. У нас уже есть шаблон, выбирающий корень (/), поэтому он поглощает весь вывод. Выход из этого затруднительного положения — делегировать согласование другим шаблонам, используя тег `apply-templates`. Следующий пример верно выводит имена всех животных в нашем документе:

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" version="1.0">
<xsl:template match="/">
Найден корневой узел!
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="animal">
Имя: <xsl:value-of select="name"/>
</xsl:template>
</xsl:stylesheet>
```

У нас все еще есть возможность добавить вывод перед и после тега `apply-templates`.

Но при вызове шаблон продолжает выбирать с текущего узла. Далее мы используем то, что успели узнать, и добавим несколько украшений.

Преобразование каталога

Только что позвонил ваш шеф, и теперь от вас требуется предоставить клиентам зоопарка доступ к его инвентаризации через Всемирную паутину, и сегодня же! После прочтения главы 15 вы должны быть полностью готовы к созданию «zoo app». Давайте начнем с создания таблицы стилей XSL, чтобы превратить наш *zooinventory.xml* в HTML:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="inventory">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="animal"/>
```

```
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="name" type="xs:string"/>
<xs:element name="animal">
<xs:complexType>
<xs:sequence>
<xs:element ref="name"/>
<xs:element name="species" type="xs:string"/>
<xs:element name="habitat" type="xs:string"/>
<xs:choice>
<xs:element name="food" type="xs:string"/>
<xs:element ref="foodRecipe"/>
</xs:choice>
<xs:element name="temperament" type="xs:string"/>
<xs:element name="weight" type="xs:double"/>
</xs:sequence>
<xs:attribute name="animalClass" default="unknown">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="unknown"/>
<xs:enumeration value="mammal"/>
<xs:enumeration value="reptile"/>
<xs:enumeration value="bird"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:element>
<xs:element name="foodRecipe">
<xs:complexType>
<xs:sequence>
<xs:element ref="name"/>
<xs:element maxOccurs="unbounded" name="ingredient" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Таблица стилей содержит три шаблона. Первый выбирает `/inventory` и выводит его в начале нашего HTML-документа (заголовок) вместе с началом таблицы для животных. Затем он делегирует, используя `apply-`

templates, перед закрытием таблицы и добавлением HTML-колонтитула. Следующий шаблон выбирает inventory/animal, выводя по одной строке в HTML-таблице для каждого животного. Хотя в документе нет других элементов animal, все же стоит определить, что мы выбираем animal только в контексте inventory, поскольку в этом случае мы связываем начало и конец нашей таблицы с inventory. Этот шаблон имеет смысл только в контексте inventory. Наконец, мы предоставляем шаблон, который выбирает foodRecipe и выводит небольшую вложенную таблицу с этой информацией. foodRecipe применяет операцию «for-each» в цикле по всем дочерним узлам, используя select, определяющий, что мы заинтересованы только в дочерних элементах ingredient. Для каждого ingredient мы выводим его значение в строку. Отметим еще кое-что, касающееся шаблона animal. Наш элемент apply-templates имеет атрибут select, ограничивающий элементы, на которые действует. В этом случае мы используем регулярный синтаксис выражения |, чтобы сказать, что мы хотим применить шаблон только к дочерним элементам food или foodRecipe. Почему мы делаем это? Поскольку мы не выбрали корень документа (только inventory), поведение таблицы стилей при выводе простого текста узлов, которые больше нигде не выбраны, все еще остается предопределенным. Мы используем это поведение, чтобы вывести текст элемента food. Но мы не хотим выводить текст остальных элементов animal, которые мы уже вывели явно, так что мы обрабатываем только элементы food и foodRecipe. Мы могли бы предпочесть более многословный вариант, добавив шаблон, выбирающий корневой элемент, и еще один шаблон, только для элемента food. Это также может означать, что новые теги, добавленные в наш XML, по умолчанию будут игнорироваться и не выводиться. Это поведение может вас устраивать, а может и нет. Также есть и другие возможности. Как и в случае со всеми мощными инструментами, существует более одного способа сделать что-либо.

Преобразование XLS

Теперь, когда у нас есть таблица, давайте применим ее! Следующая простая программа, XSLTransform, использует пакет javax.xml.transform для применения таблиц к XML-документу и вывода результата. Вы можете использовать ее для экспериментов с XSL и с кодом нашего примера.

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
public class XSLTransform
{
public static void main( String [] args ) throws Exception {
```

```
if ( args.length < 2 || !args[0].endsWith(".xsl") ) {
System.err.println("usage: XSLTransform file.xsl file.xml");
System.exit(1);
}
String xslFile = args[0], xmlFile = args[1];
TransformerFactory factory = TransformerFactory.newInstance();
Transformer transformer =
factory.newTransformer( new StreamSource( xslFile ) );
StreamSource xmlsource = new StreamSource( xmlFile );
StreamResult output = new StreamResult( System.out );
transformer.transform( xmlsource, output );
}
```

При запуске `XSLTransform`, передается таблица XSL и входящий XML, как показано в следующей команде:

```
% java XSLTransform zooinventory.xsl zooinventory.xml > zooinventory.html
```

Вывод должен выглядеть, как показано на рис. 24.2.

Zoo Inventory

Name	Species	Habitat	Temperament	Diet
Song Fang	Giant Panda	China	Friendly	Bamboo
Cocoa	Gorilla	Central Africa	Know-it-all	*Gorilla Chow* Fruit Shoots Leaves

Рис. 24.2. Вид таблицы инвентаризации зоопарка

Выполнение преобразования — процесс, похожий на работу синтаксических анализаторов SAX или DOM. Разница в том, что ранее мы использовали `TransformerFactory`, а в этот раз, мы создаем преобразователь, передавая таблицу XSL, как источник. Итоговый объект `Transformer`, затем передается машине, которая знает, как получить вывод XML и создать вывод соответствующий этим правилам.

Необходимо отметить одну важную вещь относительно XSLTransform: он не гарантирует безопасности потока. В нашем примере мы запускаем преобразование только один раз. Если вы планируете запускать одно и то же преобразование несколько раз, вы должны добавить дополнительное действие в объект `Templates` для первого преобразования, используемого далее для создания объектов `Transformer`.

```
Templates templates =
factory.newTemplates( new StreamSource( args[0] ) );
Transformer transformer = templates.newTransformer();
```

Объект `Templates` содержит синтаксически анализированное представление таблицы в компилированной форме и делает процесс получения нового `Transformer` более быстрым. Объекты `transformer` в этом случае сами могут быть более высоко оптимизированными. XSL преобразование на самом деле создает байт-код для эффективного «перевода», который реализует преобразование. Это значит, что преобразователь, вместо того чтобы читать описание, что делать с вашим XML, на самом деле предоставляет небольшую скомпилированную программу для выполнения инструкций!

XLS в браузере

На нашем примере XSLTransform вы можете понять, как вы можете поступать при визуализации XML в HTML документе на серверной стороне. Но, как упоминалось во введении, современные веб-браузеры так же хорошо поддерживают XSL на стороне клиента. Браузеры могут автоматически загружать XSL-таблицы и использовать их для преобразования в XML-документ. Чтобы это произошло, просто добавьте ссылку на стандартную XSL-таблицу в вашем XML. Вы можете вставить директиву `stylesheet` после декларации `DOCTYPE` в файле *zooinventory.xml*:

```
<?xml-stylesheet type="text/xsl" href="zooinventory.xsl"?>
```

Пока *zooinventory.xsl* файл доступен в той же директории (по основному URL-адресу), что и файл zooinventory.xml, браузер будет использовать его для визуализации HTML на стороне клиента.

Веб-сервисы

Как вы могли убедиться благодаря примерам в главах 14 и 15, одно из наиболее интересных применений XML — это веб-сервисы. Веб-сервисы — это просто сервисные приложения поддерживающие работу по сети, с применением XML для описания запросов и товаров. Обычно веб-сервисы работают по HTML и используют основанный на XML протокол Simple `Object` Access Protocol (SOAP), стандарта W3C. Комбинация XML и HTTP предоставляет широкий выбор доступных интерфейсов для сервисов.

SOAP и другие основанные на XML удаленные процедуры вызывают механизм, который может быть использован вместо Java RMI для кросс-платформенного взаимодействия. Веб-сервисы широко используются, и, похоже, их значение с годами будет расти. Чтобы узнать больше о Java API, связанных с веб-сервисами, посетите сайт **java.sun.com/webservices/**.

Это было краткое введение в XML. Вам еще многое предстоит узнать об этой захватывающей области программирования, к тому же API развиваются быстрыми темпами. Мы надеемся, что дали вам хороший старт.

Пара слов в заключение

Этой главой мы завершаем основную часть нашей книги. Мы надеемся, что она вам понравилась. Это было долгое и восхитительное путешествие по разработке Java. И по обыкновению, добавим, что мы всегда рады обратной связи — это поможет нам в дальнейшем сделать нашу книгу еще лучше.

Приложение А
СРЕДА РАЗРАБОТКИ ECLIPSE

В этой книге мы попытались сфокусироваться на языке Java и API, не уделяя много времени обсуждению специфических инструментов. Но программирование на Java сегодня действительно требует использования Интегрированных Средств Разработки (IDE). Современные языки и инструменты разработки настолько прочно переплелись, что трудно представить работу над большими проектами без поддержки IDE. Современные инструменты разработки «понимают» язык глубже и дают вам огромную силу для создания, поиска, изменения и решения проблем вашего кода.

Доступен широкий спектр Java IDE с различными функциями. Как все мощные инструменты, IDE постоянно изменяются и развиваются. Уже много лет мы отдаем предпочтение Intellij IDEA компании JetBrains. Тем не менее наиболее широко используемым IDE для Java остается проект с открытым исходным кодом Eclipse. Во введении к этой книге мы кратко представили Eclipse. В данном приложении мы пойдем немного дальше и воспользуемся Eclipse для загрузки и просмотра кода из этой книги, который собран для вас в качестве проекта Eclipse.

У IDE есть как преимущества, так и некоторые недостатки, особенно для начинающего Java-программиста. К преимуществам относится представление «все в одном» исходных кодов Java, с подсветкой синтаксиса, поиском справки, контролем проектов, встроенной документацией, сборкой, реорганизацией и разработкой — все под рукой. Исторически сложилось так, что инструменты «все в одном» стремятся превратиться в всеобъемлющий инструмент, который делает пользователя всемогущим, но абсолютно беспомощным за его пределами. IDE, как правило, поощряют чрезмерно упрощенные макеты без структуры или разделов, чтобы их было проще понять. IDE может также стать спутанным клубком состояний и информации о проекте, которую просто невозможно использовать совместно с другими разработчиками или при взаимодействии между проектами. Многие из этих проблем относятся к последнему поколению IDE, но для большинства разработчиков преимущества значительно перевешивают недостатки.

Войны IDE

Бесполезно сравнивать функции IDE по отдельности потому, что все современные IDE основаны на архитектуре подключаемых модулей, что позволяет использование предоставленных третьей стороной инструментов. Говоря, что IDE имеет функцию X, мы лишь провоцируем возражение, что IDE имеет подключаемый модуль для X и Y. Но все же стоит потратить немного времени и провести сравнение (если мы осмелимся). В этой книге мы используем редакторы NetBeans 7.2 и Eclipse 4.2. Какой же из них лучше? Краткий ответ — на момент написания этой книги Eclipse является более популярным и чуть более изысканным за счет зависимости от платформы, в то время как последняя версия NetBeans предоставляет немного более расширенные возможности за ее пределами. NetBeans предоставляет визуальный конструктор и среду разработки веб-приложений. Конечно, вы можете добавить их в Eclipse, но вам придется выбирать из ряда вариантов (иногда платных). Другая важная функция NetBeans — это то, что он использует полностью внешний процесс сборки Ant. Это означает, что вы можете создать ваше приложение как внутри, так и вне данного IDE, несомненно, похожими способами. С учетом сказанного перейдем к Eclipse.

Знакомство с Eclipse

Начнем. Во-первых, вам необходимо установить Eclipse. Скачайте последнюю версию «Eclipse IDE for Java Developers». Выберите верную версию для вашей платформы. Распакуйте ZIP-файл в выбранную вами папку и затем запустите приложение.

При первом запуске Eclipse вам необходимо будет выбрать рабочее пространство (workspace). Это корневой каталог для хранения новых проектов, которые вы создаете в Eclipse. По умолчанию каталог может находиться внутри папки, где установлено приложение, что может вас не устраивать. Выберите каталог и нажмите кнопку Ok

Eclipse встречает вас окном приветствия (Welcome screen). Закройте это окно. Если вы хотите вернуться позже и перейти к примерам Eclipse и связным темам справки, вы можете вызвать это окно вновь, выбрав команду меню **Help ⇒ Welcome** (Справка ⇒ Окно приветствия).

Перед тем как мы продолжим, отметим еще, что Eclipse хранит всю свою конфигурационную информацию в папке configuration, внутри ди-

ректории инсталляции. Если в какой-то момент вы почувствуете, что что-то неверно и вы хотите начать с нуля, вы можете выйти из приложения и удалить данную папку.

Вы можете также удалить вашу папку с проектами, они хранятся в попроектном состоянии. Если вы хотите перезагрузить все окна приложения в их изначальном положении, вы можете выбрать команду меню **Window ⇒ Reset Perspective** (Окно ⇒ Сброс Перспектива). Мы еще поговорим о перспективе позже.

Импорт учебных примеров Java

Перед тем как мы расскажем о самом IDE, давайте загрузим примеры из этой книги. Вы можете найти ZIP-файл, содержащий все примеры этой книги, по адресу **eksmo.ru/smv/Primers.zip**. Версия примеров для Eclipse называется *examples-eclipse-appa.zip*.

Откройте окно мастера импорта, выбрав команду меню **File ⇒ Import** (Файл ⇒ Импорт). Выберите пункт **General ⇒ Existing Projects Into Workspace** (Общие ⇒ Существующие проекты в рабочей области) в качестве источника и нажмите кнопку **Next** (Далее).

Установите переключатель в положение **Select Archive File** (Выбрать файл архива). Нажмите кнопку **Browse** (Обзор) и выберите файл *examples-eclipse-appa.zip*. Диалоговое окно должно выглядеть так, как показано на рис. А.1. Нажмите кнопку **Finish** (Готово).

Теперь Eclipse импортирует все файлы из архива и незамедлительно начнет собирать исходники в фоновом режиме (небольшая строка состояния внизу экрана будет отражать процесс).

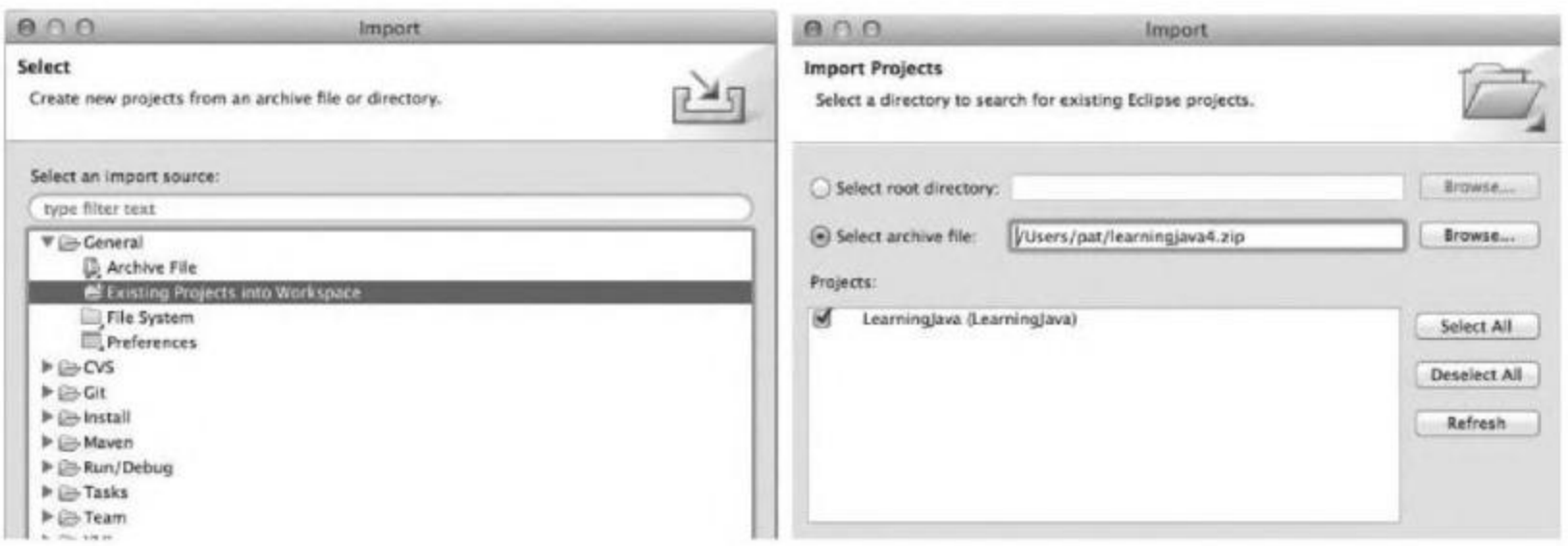

Рис. А.1. Диалоговое окно импорта проектов

Использование Eclipse

Первым делом нам необходимо настроить IDE для просмотра и редактирования исходных кодов Java. Если вы загрузили стандартную версию Java для Eclipse, она должна быть настроена для разработки Java. Если вы выбрали другой пакет, перейдите в меню **Window ⇒ Open Perspective ⇒ Java (**Окно ⇒ Открыть Перспективу ⇒ Java), чтобы перевести Eclipse в режим редактирования Java. *Перспектива* в Eclipse — это набор различных инструментов, меню и элементов управления, ориентированных на конкретные типы задач, такие как редактирование в Java или просмотр репозитория ресурсов. Вы можете открывать дополнительные инструменты и размещать элементы управления на свое усмотрение, но предустановленная перспектива поможет вам легко начать работу. Теперь примеры из этой книги отобразятся в Eclipse, как показано на рис. A.2.

В левой части окна находится проводник проектов. Здесь изображено дерево Java-пакетов, библиотек и ресурсов нашего проекта. Нажмите левой кнопкой мыши на папке, чтобы раскрыть ее и увидеть папки с исходными файлами для каждой главы книги.

В нижней части окна размещены панели, необходимые для редактирования кода на языке Java. Панель, которая раскрыта на рис. A.2, **Problems** (Задачи), показывает ошибки и предупреждения, связанные с кодом нашего проекта. Eclipse уже скомпилировал код в фоновом режиме. Как правило, вам нет необходимости сообщать ему о выполнении компиляции. Вы также заметите красный значок × на некоторых папках и исходных файлах. В этих файлах есть ошибки. Далее мы поговорим о том, почему отмечены некоторые из наших примеров. Другие панели, **Javadoc** и **Declaration** (Объявление), предоставляют информацию о файле, который мы редактируем, или о выделенном участке исходного кода. Вкладка **Declaration** (Объявление) отображает исходный код элемента, выбранного в основном окне программы.

Чтобы скрыть некоторые из ошибок, мы должны быть уверены, что Eclipse находится в режиме совместимости с Java 7.0. Откройте меню **Eclipse** (или **Window** (Окно)) и выберите команду **Preferences ⇒ Java ⇒ Compiler** (Настройки ⇒ Java ⇒ Компилятор). Затем присвойте параметру **Compiler** (Компилятор) значение **Compliance Level to 7.0** (Уровень совместимости с 7.0). Нажмите кнопку **OK**, а затем — **Yes** (Да), чтобы пересобрать исходники. (Также возможно установить уровень компилятора на пред-проектный, изменив настройки проекта.)

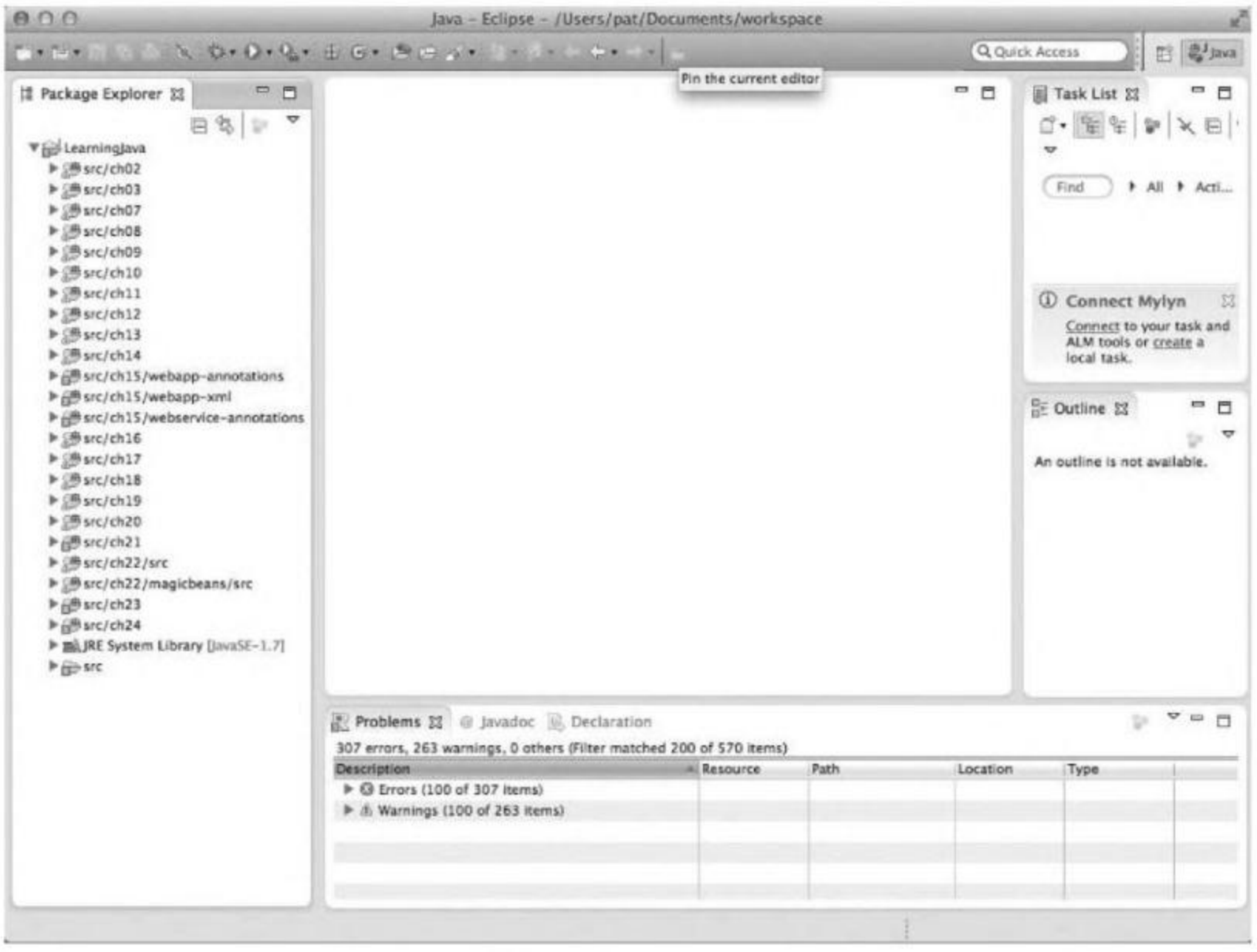

Рис. А.2. Примеры книги в Eclipse

Если некоторые ошибки в левой части программы все еще отображаются, дважды щелкните мышью по файлу *README-Eclipse.txt* в дереве проектов, чтобы прочитать справочные сведения об используемой версии программы. Некоторые из средств решения проблем связаны с генерацией исходного кода или установкой дополнительных библиотек.

Создание исходников

Давайте перейдем к исходному файлу. Перейдите к файлу *Calculator.java*, размещенному в пакете *ch19/default*. Дважды щелкните мышью, чтобы открыть исходный код, как показано на рис. А.3.

Когда вы развернули дерево исходных папок, вы можете перейти глубже уровня исходных кодов Java, раскрывая элементы файлов. Перемещаясь внутри исходного файла в проводнике, вы можете видеть части кода Java, такие, как методы, переменные и вложенные классы. Если дважды щелкнуть мышью по этим элементам, вы можете открыть файл и переместиться точно на соответствующую строку. Это похоже на представление в окне вывода, которое мы рассмотрим в дальнейшем.

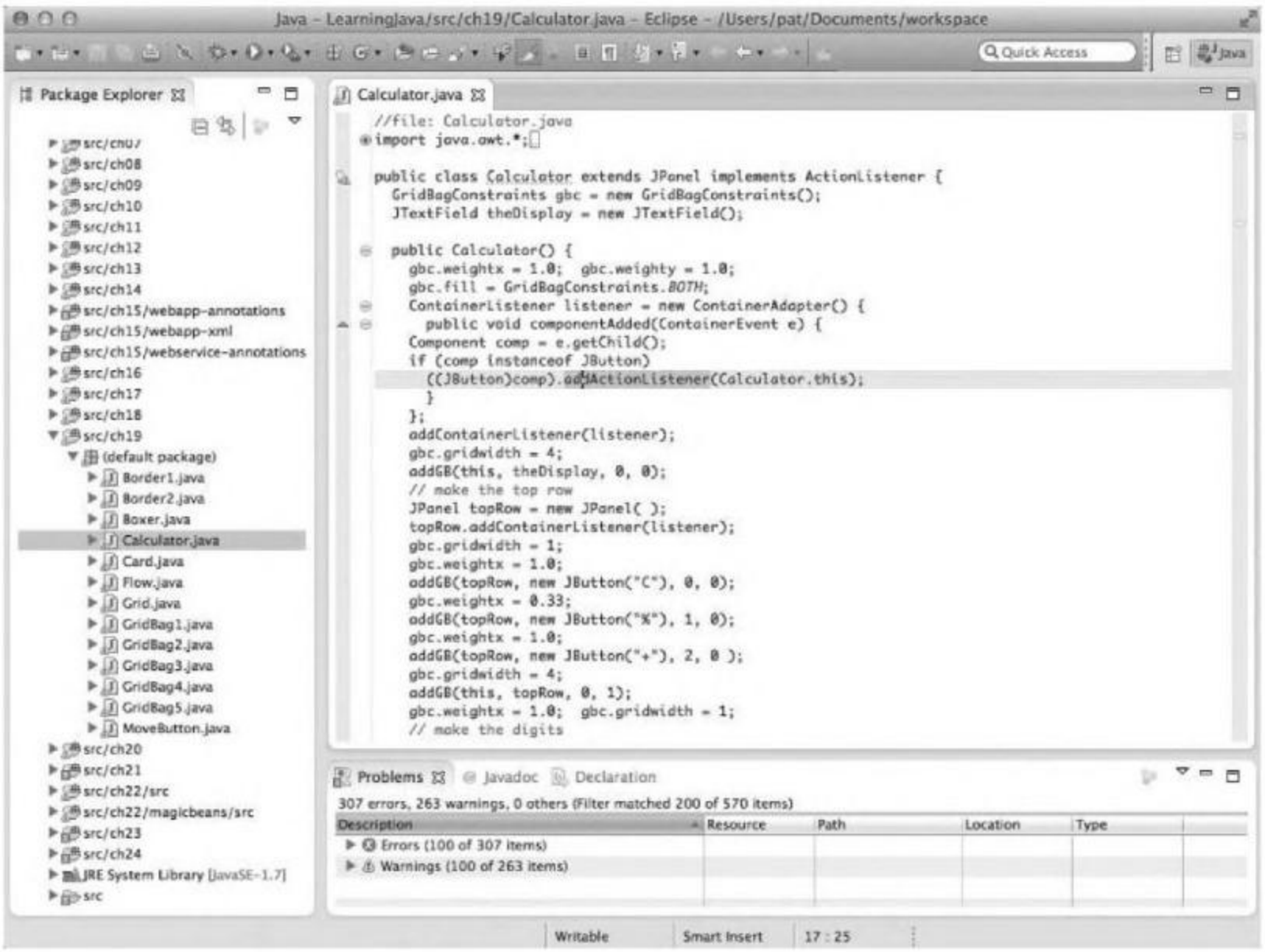

Рис. А.3. Открытие *Calculator.java*

Интерфейс

В центральной части экрана открытый файл отражается в новой вкладке редактора. Редактор, конечно, это область, где выполняется большая часть действий в Java IDE. Обратите внимание на подсветку синтаксиса цветом. Также обратите внимание, на небольшие стрелки, расположенные непосредственно слева от исходного кода. Это складки. *Складки* — это группы сегментов Java-кода, такие как метод или комментарий, которые позволяют вам скрывать или раскрывать их вывод. К примеру, операторы импорта файлов сгруппированы по умолчанию. Щелкните мышью по голубой стрелке слева от строки импорта, чтобы раскрыть их и увидеть все строки импорта.

Далее, ошибки и предупреждения подсвечиваются в столбце слева от столбца складок. Попробуйте допустить небольшую ошибку в файле. К примеру, измените импорт `javax.swing` на импорт `javax.boofa` и посмотрите, что произойдет. Не все красные символы × указывают на проблемы. Когда вы наводите указатель мыши на красный символ ×, вы увидите всплывающее сообщение с отчетом об ошибке. Столбец,

расположенный у правой полосы прокрутки, отображает местонахождение ошибок и предупреждений. Щелкая мышью в этой области, вы переходите к строке кода. Область ошибок в нижней части окна также интерактивна. Щелкните мышью по отчету о проблеме, чтобы открыть соответствующий файл на нужной строке.

В правой части окна IDE расположена панель вывода, которая отражает структурное представление Java кода, похожее на то, которое мы наблюдали при раскрытии вкладки с исходными файлами в обозревателе проектов. Щелкая мышью по полю, методу или внутреннему классу, вы перемещаетесь к соответствующей строке в редакторе кода. Элементы управления в верхней части панели позволяют фильтровать содержимое для быстрого доступа.

Запуск примеров

Чтобы запустить пример Calculator, вы можете открыть исходный код и нажать большую зеленую стрелку — кнопку **Run** (Выполнить) или выбрать файл с исходным кодом в проводнике и выбрать команду меню **Run ⇒ Run As ⇒ Java Application** (Выполнить ⇒ Выполнить как ⇒ Приложение Java). Приложение Calculator будет запущено, как показано на рис. А.4.

Вы можете использовать меню **Run As** (Выполнить как), чтобы сообщить Eclipse, что данный файл представляет собой отдельный класс с методом `main()`. Обычно в проекте много кода и один или несколько методов `main()`. В таком случае, меню **Run** (Выполнить) (и большая зеленая стрелка на панели инструментов) может быть сконфигурировано для запуска классов по умолчанию по всему приложению. Отдельный файл тоже можно запустить, используя контекстное меню, которое появится, если вы щелкнете правой кнопкой мыши по файлу (или щелкнете мышью, нажав и удерживая клавишу **Ctrl** (OS X)). Если вы выбрали приложение, которое осуществляет вывод в `System.out`, Eclipse откроет окно консоли в нижней части окна, чтобы перехватить вывод. Чтобы увидеть это в действии, запустите пример *ch10/PrintfExamples.java*. Для запуска примера, которому требуются аргументы, вы должны выбрать в меню **Run** (Выполнить) одноименную команду. Она вызовет диалоговое окно, используя которое, вы можете применять различные параметры, в том числе аргументы командной строки. Меню **Run** (Выполнить) так же включает функцию **Run History** (История Выполнения) с историей выполненных запусков.

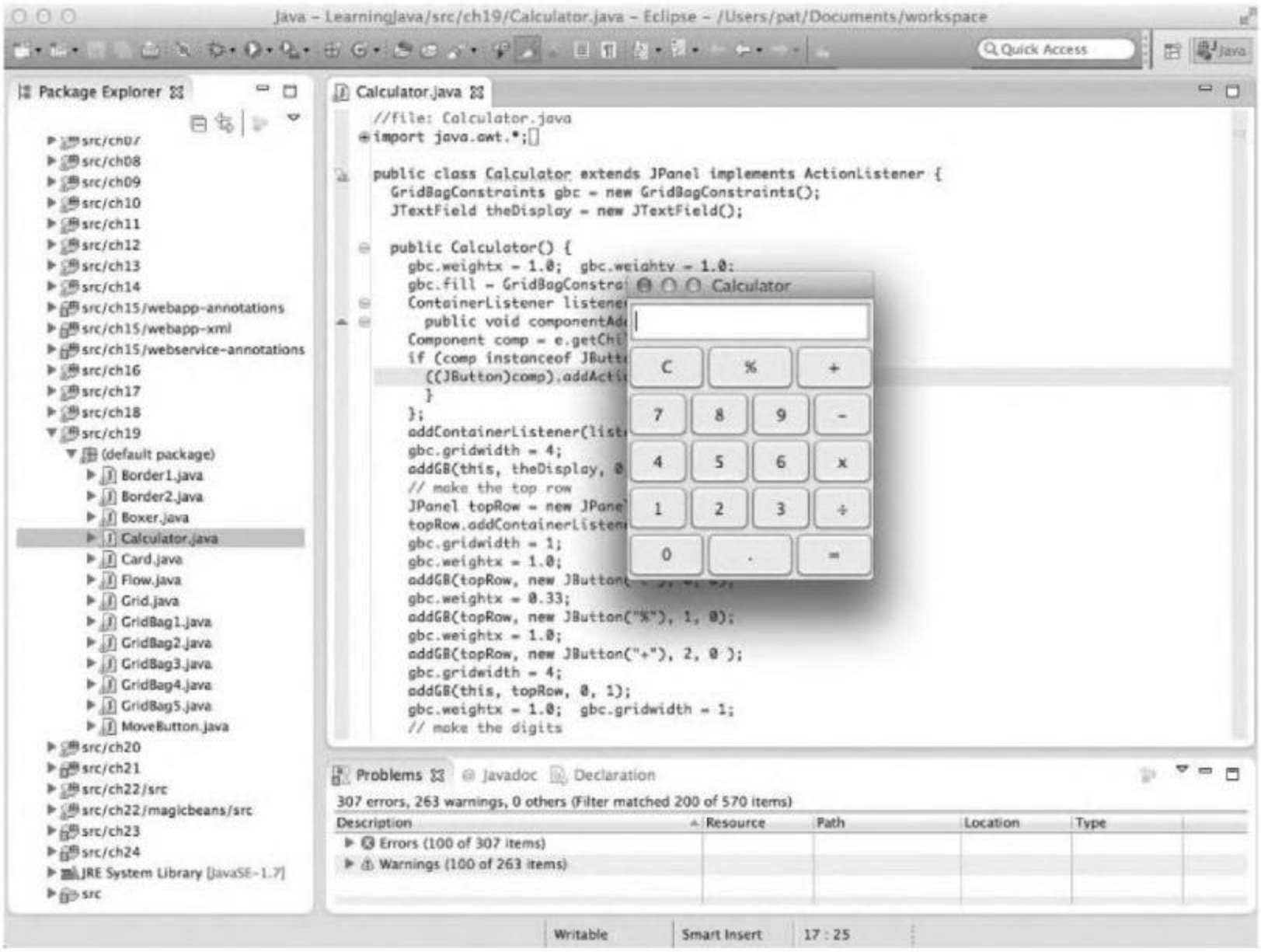

Рис. А.4. Запуск приложения Calculator

Сборка примеров в Ant-Based

Некоторые главы включают компоненты, которые должны быть собраны с поддержкой Ant Build файлов. Например, JavaBeans JAR в главе 22 и файл Web Application WAR в главе 15 имеют Ant-файлы. Вы можете запустить Ant из Eclipse, щелкнув правой кнопкой мыши по файлу *build.xml*, и выбрав из контекстного меню команду **Run As** ⇒ **Ant Build** (Выполнить как ⇒ Ant Build). Для доступа к данной возможности вы должны открыть обозреватель проектов, используя команду меню **Windows** ⇒ **Show View** ⇒ **Project Explorer** (Окна ⇒ Вид ⇒ Обозреватель проектов). (Обозреватель проектов отображает все файлы, в отличие от того, как показана иерархия Java-классов в Обозревателе пакетов.) Существует два варианта Ant Build. Первый запускает сборку по умолчанию, в то время как второй предлагает вам выбрать цель (по аналогии с командами **Run As** (Выполнить как) и **Run** (Выполнить) в одноименном меню). Например, чтобы собрать файл *magicbeans.jar* из главы 22, перейдите к файлу *sr/ch22/magicbeans/build.xml*, щелкните по нему правой кнопкой мыши и выберите команду **Run As** ⇒ **Ant Build** (Выполнить как ⇒ Ant Build). Используйте команду меню **File** ⇒ **Refresh**

(Файл ⇒ Обновить), чтобы увидеть новые файлы, которые были созданы компоновщиком Ant, в дереве проекта.

Отдельные примеры

Некоторые примеры в этой книге могут быть выполнены вне Eclipse. Например, некоторые из сетевых примеров проще запускать из командной строки. Нет объективных причин, чтобы постоянно пользоваться IDE. Вы должны уверенно использовать оба средства.

Функции Eclipse

В Eclipse есть много интересных функций, и мы продемонстрируем некоторые из них.

Создание ярлыков

Конечно, вы заметили оформление цветом кода и метаданных, когда просматривали Java-код в Eclipse.

Вы можете настроить интерфейс на свой вкус, используя команду меню **Eclipse ⇒ Preferences ⇒ Java ⇒ Editor ⇒ Syntax Coloring panel** (Eclipse ⇒ Настройки ⇒ Java ⇒ Редатор ⇒ Панель настройки подсветки синтаксиса). Многие другие функции можно настроить с помощью команды меню **Eclipse ⇒ Preferences ⇒ Java ⇒ Code Style and Editor panes** (Eclipse ⇒ Настройки ⇒ Java ⇒ Стиль кода и панель редактора).

Во время ввода кода попробуйте нажать сочетание клавиш **Ctrl+Пробел**, чтобы использовать функцию автозавершения или для отображения вариантов IDE Eclipse. Например, вместо того чтобы полностью набирать `JButton`, попробуйте ввести `JBu` и нажать сочетание клавиш **Ctrl+Пробел**. Eclipse завершит имя автоматически. В случае, если существует несколько вариантов завершения, появится список, примерно такой, как показано на рис. А.5

Eclipse также поддерживает аббревиатуры для стандартных имен. Попробуйте набрать `sysout` и нажать **Ctrl+Пробел**. Eclipse преобразует введенный текст в `System.out.println()` и автоматически поместит курсор в данный шаблон. Если вы используете кавычки, Eclipse подставит

завершающую кавычку автоматически. Обратите внимание на зеленое поле, которое размещается после закрывающей кавычки. Нажмите клавишу **Tab**, чтобы переместиться в эту область, не используя клавиши со стрелками. Нажмите клавишу **Tab** еще раз, и вы перейдете к следующей зеленой области, которая находится в конце этой строки.

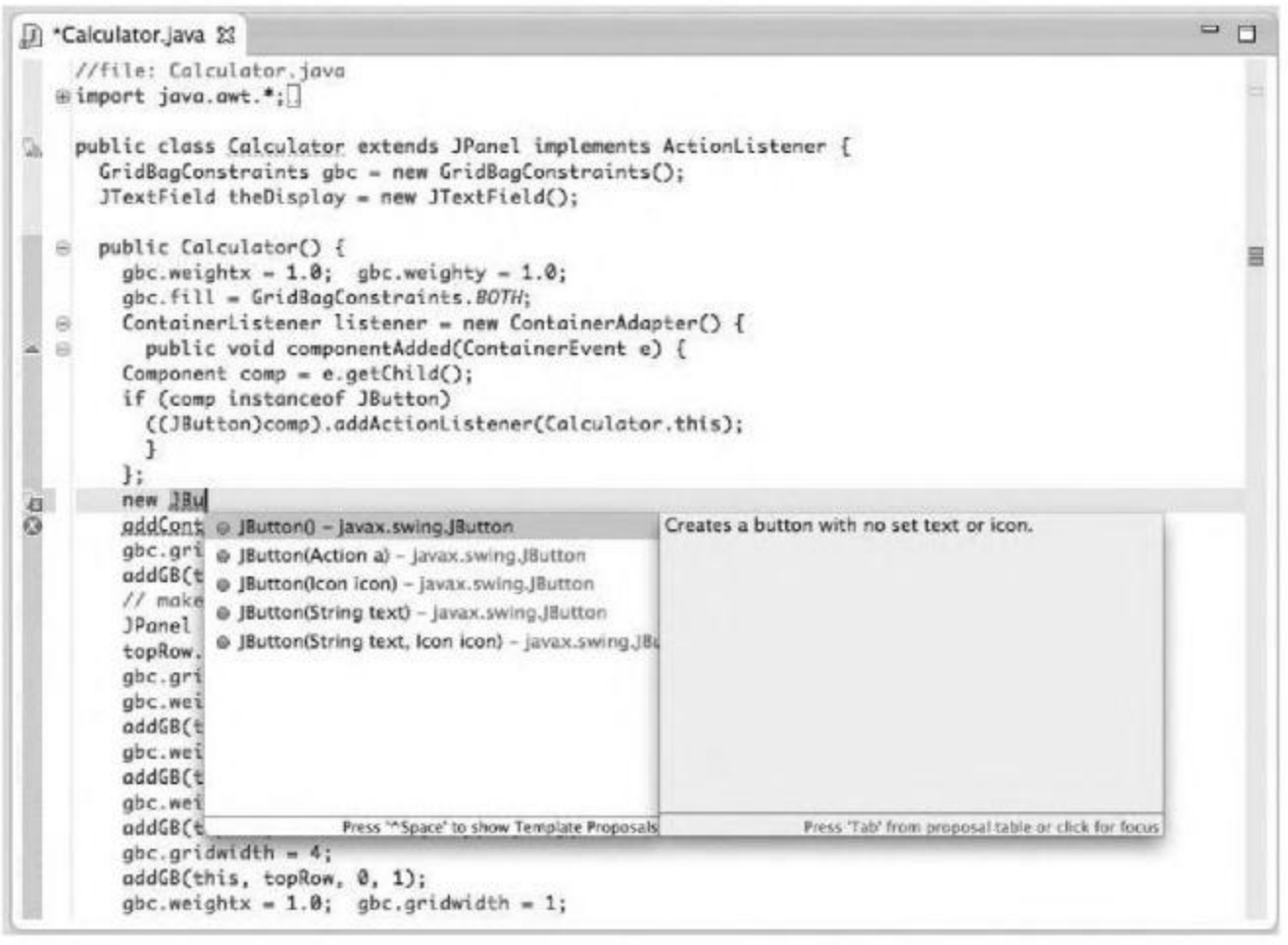

Рис. А.5. Функция автозавершения в Eclipse

Автоматическая коррекция

Eclipse предлагает исправлять простые ошибки в вводимом коде, когда обнаруживает их. Чтобы увидеть предлагаемые исправления, щелкните мышью по красному символу ×, расположенному слева от строки с ошибкой. Eclipse отобразит раскрывающийся список с возможными вариантами исправления проблемы. Просмотрите их, прежде чем вносить изменения.

К примеру, попробуйте изменить имя одного из компонентов `JButtons` в приложении Calculator на `JBBut`. Теперь щелкните мышью по красному символу ×, и вы увидите нечто похожее на рис. А.6. Eclipse предлагает несколько возможных исправлений, лучший из которых — исправить опечатку и изменить имя на `JButton`. Конечно, если вы действительно хотите создать новый тип кнопки, то могли выбрать вариант **Create class** (Создать класс) и Eclipse помог бы создать его, предоставив шаблон.

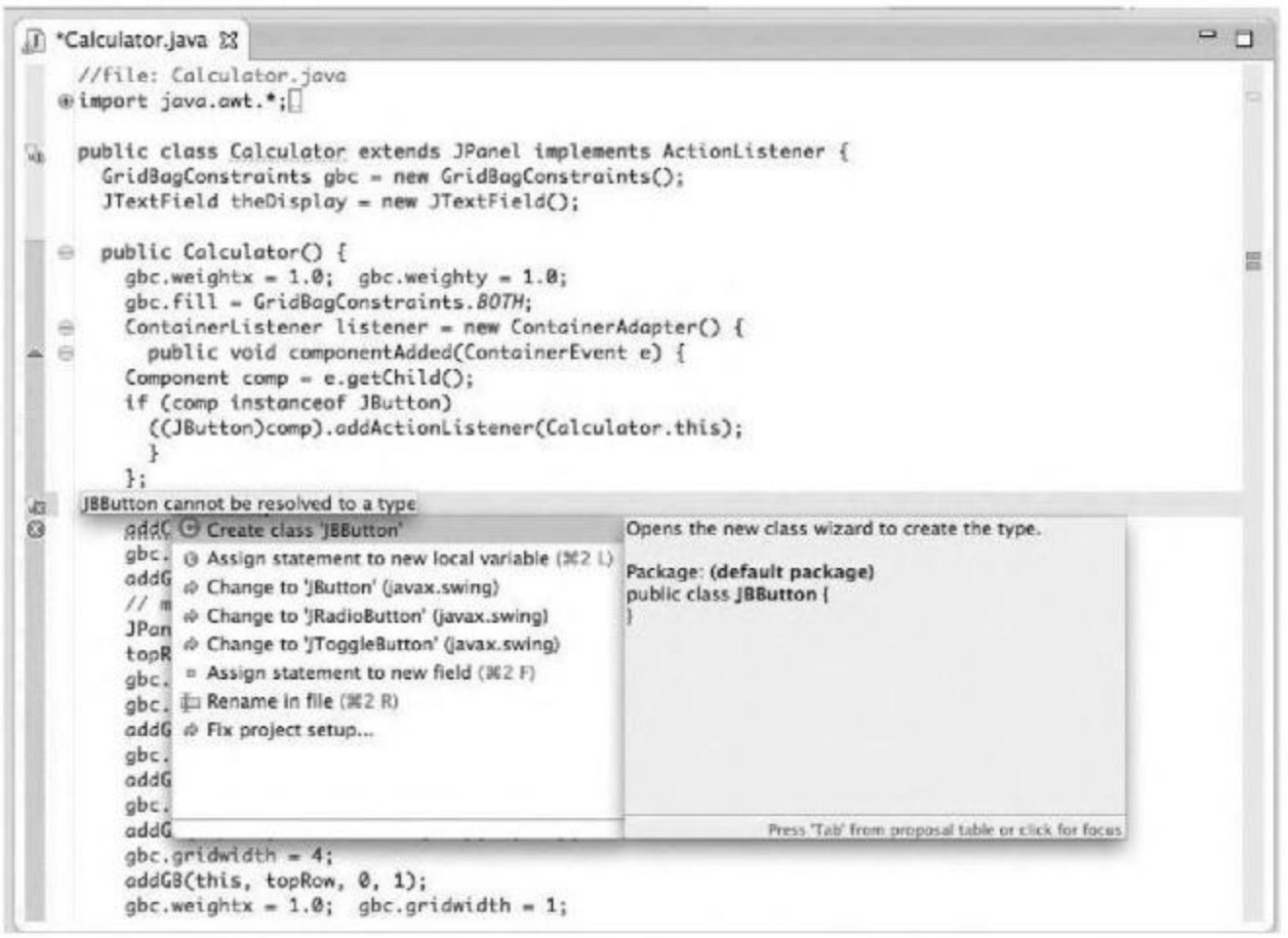

Рис. А.6. Функция автокоррекции в Eclipse

Реорганизация кода

Eclipse предлагает набор инструментов под единым заголовком *реорганизация кода*. Они включают возможность переименования и перемещения элементов, автоматическое смещение ссылок на них и изменение их по всему проекту.

Также доступны возможности создания интерфейса для классов, копирования всех открытых методов или добавления шаблонов в ваш код, чтобы инкапсулировать все объекты, созданные для типа. Вы даже можете инкапсулировать доступ к переменным, изменяя код в обход использования метода доступа. Эти инструменты помогут сэкономить время, уходящее на ввод кода вручную.

Например, взгляните еще раз на приложение Calculator. Щелкните мышью по методу `addGB()` в выводе или выберите любой другой метод. Мы часто используем этот метод, поэтому присвоим ему более подходящее имя. Выберите команду меню **Refactor** ⇒ **Rename** (Реорганизация кода ⇒ Переименовать) и измените имя метода на `addToGridBag`. Если вы хотите просмотреть изменения, нажмите кнопку **Preview** (Предпросмотр), чтобы увидеть их.

Затем нажмите кнопку **OK**, после чего Eclipse изменит имя и все ссылки на метод в вашем проекте (в данном случае, изменения должны

коснуться всех директорий с исходным кодом). Вы также можете произвести реорганизацию кода методов в части изменения их сигнатуры и добавления или удаления аргументов.

Сравнение файлов

Eclipse предоставляет возможность быстрого сравнения двух классов. Выберите два файла в обозревателе пакетов и щелкните правой кнопкой мыши по одному из них. В появившемся контекстном меню выберите команду **Compare With ⇒ Each Other** (Сравнить с ⇒ Друг с другом).

Eclipse откроет панель просмотра двух кодов, которая отобразит разницу между файлами в графическом представлении. Вы можете решать проблемы простым копированием фрагментов кода из одного файла в другой, используя стрелки в верхней части панели. Например, откройте файлы *ch15/webapp-xml/all-web.xml и ch15/webapp-xml/filters-web.xml и* выявите различия. Вы также можете увидеть разницу после внесения изменений, выполнив команду меню **Compare With ⇒ Local History** (Сравнить с ⇒ История изменений), — данная команда позволит вам сравнить текущее состояние файла с предыдущим, сохраненным в проекте. Это очень полезный инструмент.

Организация импорта

Eclipse может приводить в порядок операторы импорта в вашем исходном коде. Команда меню **Source ⇒ Organize Imports** (Источник ⇒ Организовать импорт) инструктирует Eclipse объединить пакеты импорта в единственный класс импорта. Eclipse автоматически определяет, какие классы и из какого пакета используются, а затем объединяет пакеты импорта в отдельные импорты. Это делает код немного более явным, но некоторые при использовании большого числа классов предпочитают пакетный импорт.

Форматирование исходного кода

Eclipse поддерживает автоматическое форматирование исходного кода. Для этого используйте команду меню **Source ⇒ Format option** (Источник ⇒ Опции форматирования). Стиль форматирования настраивается с помощью команды меню **Window ⇒ Preferences ⇒ Java ⇒ Code**

Style ⇒ **Formatter** (Окно ⇒ Настройки ⇒ Java ⇒ Стиль кода ⇒ Форматирование).

Заключение

Это приложение представляет собой краткое введение в возможности Eclipse и других Java IDE. Вы можете самостоятельно узнать больше о том, как с помощью таких инструментов повысить вашу производительность как разработчика Java. Также советуем сравнить Eclipse, NetBean и другие варианты, такие как Intellij IDEA, чтобы выбрать наиболее подходящий вам. И не забудете почитать об Apache Ant для компиляции Java-приложений.

Приложение Б

BEANSHELL: НАПИСАНИЕ СЦЕНАРИЕВ JAVA

В этой книге мы (в данном случае я, Пат) избегали обсуждения большинства сторонних инструментов, которые не являются частью JDK. Я готов сделать исключение, чтобы рассказать об изящном, бесплатном инструменте Java, называемом BeanShell. Как видно из названия, BeanShell может быть использован в качестве «оболочки» Java. Этот инструмент дает вам возможность использовать стандартный синтаксис Java — операторы, выражения и даже классы — в командной строке или оценивать биты исходного кода в вашем приложении. С BeanShell вы можете опробовать кусочки кода, пока работаете с книгой. Вы можете получить доступ ко всем Java API и даже создать компоненты графического пользовательского интерфейса и управлять ими в режиме реального времени. BeanShell использует только рефлексию, поэтому нет необходимости компилировать файлы классов.

Я работал в BeanShell, пока разрабатывал примеры для данной книги, и думаю, BeanShell может стать веселым компаньоном в вашем путешествии по Java. BeanShell — это программное обеспечение с открытым исходным кодом и, пока пишется эта книга, происходит процесс адаптации BeanShell под Apache. Вы можете найти последнее обновление и подробную информацию на домашней странице проекта (**www.beanshell.org**). На протяжении нескольких лет BeanShell остается популярным средством и широко используется в инструментах и проектах, которым необходимы сценарии. Однако на сегодняшний день, возможно, существуют лучшие средства для написания серьезных Java-скриптов. Например, вы можете рассмотреть Jython, реализацию языка сценариев Python, как второй вариант. Многие языки сценариев могут получать доступ к `java.scripting` API (JSR-223), используя плагины. JDK 7 поставляется совместно с Mozilla Rhino реализацией JavaScript.

Запуск BeanShell

Все, что необходимо для запуска BeanShell — это среда выполнения Java (версии 1.1 или более поздней) и *bsh* файл JAR. В системах OS X

и Windows, вы можете запустить графическое настольное приложение, дважды щелкнув мышью по JAR-файлу. В более общем случае вы можете добавить JAR в директорию classpath.

```
Unix: export CLASSPATH=$CLASSPATH:bsh.jar
Windows: set classpath %classpath%;bsh.jar
```

В последующем вы сможете запускать BeanShell интерактивно в GUI или в режиме командной строки.

```
java bsh.Console // запускает графическое приложение
java bsh.Interpreter // запускает в текстовом режиме в командной строке
```

Запуск BeanShell с консолью GUI организует простое Swing-приложение, позволяющее открывать несколько окон оболочки, и допускает использование возможностей истории команд, редактирования строк и вырезания/вставки. Существует еще несколько средств GUI столь же доступных, включающих простой текстовый редактор и проводник классов. В качестве альтернативы вы можете запустить BeanShell в текстовом виде в командной строке.

Вы можете запускать сценарии BeanShell из файлов следующим образом:

```
% java bsh.Interpreter myfile.bsh
```

В некоторых версиях NetBeans и Sun Java Studio IDE вы можете создавать файлы сценариев BeanShell, используя команду **New File** (Новый файл) или запуская любой файл с расширением *bsh*, как если бы вы хотели выполнить Java-код.

Операторы и выражения Java

В командной строке или в сценариях beanshell вы можете использовать стандартные в Java-операторы и выражения. Операторы и выражения все те же, что вы включаете в Java-метод: объявление и присвоение переменных, вызов методов, циклы и условия. Вы можете объявлять классы привычным способом, но BeanShell также позволяет вам записывать выражения вне классов или записывать методы в неструктурированном виде.

Вы можете писать выражения точно так же, как они должны выглядеть в Java. Также у вас есть возможность работать в стиле многих язы-

ков сценариев, используя переменные и аргументы не указывая их тип. Другими словами, вы можете не объявлять типы переменных, которые используете (как примитивы, так и объекты).

BeanShell выдаст ошибку, если вы ошибочно примените переменную в текущем контексте. Если вы объявите типы переменных, BeanShell будет соблюдать их.

Несколько примеров:

```
foo = "Нечто";
four = (2 + 2)*2/2;
print( Нечто + " = " + Четыре ); // print() - bsh команда
// выполнение цикла
for (i=0; i<5; i++)
   print(i);
// Вывод AWT фрейма с кнопкой
button = new JButton("Моя кнопка ");
frame = new JFrame("Мой фрейм");
frame.getContentPane().add( button, "Центр" );
frame.pack();
frame.setVisible( true );
```

Если вы не приветствуете идею «свободного» синтаксиса Java, то можете отключить данную функцию в BeanShell следующей командой `setStrictJava( true );`

Импорт

По умолчанию, BeanShell автоматически импортирует все основные пакеты Java. Вы можете импортировать собственные классы, используя стандартное объявление импорта в Java.

```
import mypackage.*;
```

В дополнение к обычным пакетам, классам и статическому импорту, BeanShell может импортировать методы и переменные из экземпляров объекта в текущий контекст, используя команду: `importObject()`.

Пример:

```
Map map = new HashMap(); importObject( map ); put("нечто", "панель");
print( get("нечто") ); // "панель"
```

Команды BeanShell

В BeanShell присутствет набор полезных встроенных команд, представленных в виде Java методов. Эти команды реализованы в виде сценариев BeanShell и приложены в виде *bsh* файлов JAR. Вы можете создавать собственные команды, определяя методы в собственных сценариях или добавляя их в вашу директорию classpath. Для получения более подробной информации обратитесь к руководству по BeanShell.

Одна из важных команд BeanShell — `print()`, которая выводит значения. Команда `print()` в достаточной мере схожа с командой `System.out.println()`, за исключением того, что поддерживает вывод только в командную строку (если у вас открыто несколько окон). Кроме того, команда `print()` выводит некоторые типы объектов (например, массивы) более подробно, чем Java. Другая часто используемая команда — `show()`, которая включает или отключает автоматический вывод результатов каждой строки ввода (вы можете включить ее, если хотите видеть каждое вводимое значение).

Ниже представлено несколько примеров других команд BeanShell:

`source(), run()`

Считывает сценарий в данный интерпретатор или запускает в новом

`frame()`

Отображает компоненты AWT или Swing во фрейм

`load(), save()`

Загружает или сохраняет сериализуемые объекты (такие как JavaBeans)

`cd()`
`cat()`
`dir()`
`pwd()`

Unix –подобные shell-команды

`exec()`

Запускает собственные приложения

```
addClassPath()
reloadClasses()
```

Изменяют директорию classpath или перезагружают классы

```
javap()
```

Выводит классы в стиле javap определения для классов или специфичных объектов.

Полный список команд смотрите в руководстве пользователя BeanShell.

Сценарии методов и объектов

Вы можете использовать методы и объекты в BeanShell, так же, как используете их в Java-классах.

```
int addTwoNumbers( int a, int b ) {
   return a + b;
}
sum = addTwoNumbers( 5, 7 ); // 12
```

BeanShell-методы могут иметь динамические (свободные) аргументы и возвращать типы.

```
add( a, b ) {
   return a + b;
}
foo = add(1, 2); // 3
foo = add("Привет ", "Котенок"); // "Привет Котенок"
```

В BeanShell, так же как в JavaScript и Perl, методы *завершения* могут занять место классов для сценариев объектов (но в BeanShell вы также можете использовать регулярный синтаксис классов). Вы можете включить контекст вызова метода в ссылку на объект. В дальнейшем вы можете использовать эти ссылки для определения любых переменных, которые были установлены во время вызова метода. Чтобы быть полезным, объекту также нужны методы, так в BeanShell методы могут быть любого уровня вложенности. Вот простой пример:

```
user( n ) {
   name = n;
```

```
    reset() {
        print( "Перегрузка пользователя:"+name );
    }
    return this; // Вернуть объект user
}
bob = user("Боб" );
print( bob.name ); // "Боб"
bob.reset(); // вывести "Перезагрузка пользователя: Боб"
```

Этот пример присваивает содержимое метода `user()` переменной `bob` и относится к полю `bob.name` и методу `bob.reset`.

Если это кажется вам странным, не беспокойтесь. Наиболее распространенная причина желания создать сценарий объекта — реализация Java-интерфейса, и вы можете использовать стандартный анонимный синтаксис внутренних классов Java, как будет рассмотрено далее, или использовать регулярные классы. BeanShell предоставляет вам множество вариантов.

Сценарии интерфейсов и адаптеров

Одна из наиболее сильных функций BeanShell заключается в том, что вы можете написать сценарий для любого типа интерфейса. Сценарии BeanShell для объектов могут автоматически реализовывать любые запрашиваемые интерфейсы.

Единственное, что вам необходимо сделать — реализовать необходимый метод (или единственный, который будет вызван). Вы можете использовать эту функцию или как явную ссылку на BeanShell-сценарий, используя `this`-ссылку, как описывалось ранее, или используя стандартный анонимный синтаксис внутренних классов Java. Пример:

```
actionPerformed( event ) { print( event ); }
button = new JButton("Нажми меня!");
button.addActionListener( this );
frame( button );
```

Вы можете набрать этот код прямо в командной строке и нажать кнопку, чтобы увидеть созданное событие. В этом случае `this`-ссылка ссылается на текущий контекст, как на метод. BeanShell автоматически вызовет интерфейс `ActionListener` и передаст ему вызов от метода `actionPerformed()` к вашему методу в сценарии.

Или же мы можем использовать анонимный внутренний синтаксис классов, чтобы создать интерфейс `ActionListener` для нашей кнопки:

```
button = new JButton("Нажми меня!");
button.addActionListener( new ActionListener() {
   actionPerformed( event ) { print( event ); }
} );
frame( button );
```

В этом случае, «анонимный внутренний класс» — на самом деле сценарий BeanShell, который автоматически вызывает интерфейс `Action Listener` способом, похожим на тот, что использовался в предыдущем примере.

Еще кое-что: вы можете реализовывать только те методы интерфейсов, которые хотите использовать. Если вы не создавали сценарии метода, это нормально, пока он не вызывается (в этом случае вы получите исключение).

Для удобства реализации больших интерфейсов вы можете определить специальный метод `invoke()`, который автоматически реализует несуществующие в сценарии методы:

```
invoke( name, args ) { print("Метод: "+name+" вызван!"); }
```

Здесь метод `invoke()` будет автоматически реализовывать методы, которые не были определены, и просто выводить их имена. См. руководство пользователя для получения подробной информации.

Изменение директории classpath

В BeanShell, вы можете добавить к вашей директории classpath и даже перегрузить классы.

```
addClassPath("mystuff.jar");
addClassPath("http://examples.oreilly.com/learnjava3/magicbeans.jar");
```

Для перезагрузки всех классов в директории classpath используйте команду:

```
reloadClasses();
```

Вы можете делать более сложные вещи, такие как перезагрузка отдельных классов, если знаете что делаете. См. руководство пользователя для получения подробной информации

Дополнительные сведения

В BeanShell намного больше функций, чем описано здесь. Вы можете добавлять BeanShell в ваше приложение как простое средство создания сценариев, в реальном времени передавая Java объекты в скрипты и из скриптов. Вы даже можете запустить BeanShell в режиме удаленного доступа к серверу, что позволит вам работать в оболочке внутри вашего запущенного приложения, для отладки и тестирования. Также существуют BeanShell-сервлеты, которые можно использовать для запуска скриптов внутри серверных приложений.

BeanShell — небольшой бесплатный инструмент (всего около 200 Кб), обладающий лицензией открытого программного обеспечения. Вы можете узнать о нем больше, ознакомившись с полным руководством пользователя и разделом FAQ на веб-сайте.

ГЛОССАРИЙ

<applet>

HTML-тег, с помощью которого апплеты встраиваются в веб-страницу.

<object>

Тег языка HTML, с помощью которого в веб-страницы встраиваются мультимедийные объекты и приложения, такие как апплеты.

abstract

Ключевое слово `abstract` используется для объявления абстрактных методов и классов. У абстрактного метода нет реализации; он содержит список аргумент и тип возвращаемого значения, но вместо тела, заключенного в фигурные скобки, указывается точка с запятой. Реализация абстрактного метода предоставляется производной того класса, в котором он был объявлен. Само наличие абстрактного метода делает класс абстрактным.

Ant

Популярный инструмент для сборки приложений на языке Java, основанный на формате XML. Может компилировать, упаковывать и развертывать исходный код, попутно генерируя документацию. С помощью подключаемых «целей» может выполнять и другие задачи.

API (Application Programming Interface)

API-интерфейс состоит из методов и переменных, которые используются для работы с различными компонентами и вспомогательными классами. API-интерфейсы языка Java содержат классы и методы, предоставляемые пакетами `java.lang`, `java.util`, `java.io`, `java.text`, `java.net` и многими другими.

API-интерфейс для ведения журнала

API-интерфейс для ведения структурированного журнала и обмена сообщениями между программными компонентами. Поддерживает разные уровни, отражающие важность тех или иных сообщений; имеет средства фильтрации и вывода.

API-интерфейс для работы с коллекциями

Классы, которые находятся в стандартном пакете `java.util` и предназначены для работы со структурированными коллекциями или ассоциативными массивами. Этот API-интерфейс поддерживает операции сортировки и включает в себя как старые классы `Vector` и `Hashtable`, так и более новые компоненты: `List`, `Map` и `Queue`.

API-интерфейс для работы с регулярными выражениями

Пакет `java.util.regex`; предназначен для работы с регулярными выражениями. Может использоваться для выполнения поиска и замены участков текста с учетом сложных шаблонов.

appletviewer

Приложение от компании Sun, которое выполняет и отображает апплеты за пределами веб-браузера.

apt (Annotation Processing Tool)

Надстройка над компилятором для языка Java, которая генерирует аннотации при помощи подключаемых фабричных классов. Позволяет программистам реализовывать нестандартные аннотации, которые обрабатываются на уровне компилятора.

assert

Ключевое слово, с помощью которого проверяют обязательные для выполнения (согласно программной логике) условия.

Если условие, помеченное этим модификатором, возвращает `false`, генерируется ошибка, которую нельзя перехватить. Чтобы повысить производительность развертываемого приложения, этот инструмент можно отключить.

AWT (Abstract Window Toolkit)
Исходная кроссплатформенная оконная библиотека пользовательского графического интерфейса, написанная на языке Java.

BeanShell
Открытый, компактный скриптовый язык программирования, совместимый с платформой Java. Его можно использовать для прототипирования, расширения, конфигурирования и отладки приложений, а также в учебных целях.

Boolean
Простой тип данных в языке Java. Может иметь одно из двух значений: `true` или `false`.

byte
Простой тип данных в языке Java, который представляет в дополнительном коде 8-битное число со знаком.

catch
Оператор `catch` находится в начале блока для обработки исключений, сразу за блоком `try`. После ключевого слова `catch` нужно указать тип исключения, имя аргумента в круглых скобках и блок кода в фигурных скобках.

char
Простой тип данных в языке Java. Используется для хранения 16-битных символов в кодировке Unicode.

classpath
Набор путей, ведущих к определенным директориям и архивам, внутри которых содержатся скомпилированные классы и ресурсы. Используется для поиска программных компонентов.

DOM (Document Object Model)
Полноценное представление XML-документа, находящееся в памяти. Состоит из таких объектов, как `Element`, `Attribute` и `Text`. Привязка XML DOM API, доступная на платформе Java и стандартизированная консорциумом W3C.

double
Простой тип данных в языке Java. Используется для хранения 64-битных чисел с плавающей запятой.

DTD (Document Type Definition)
Специализированный язык, который описывает ограниченный набор тегов и атрибутов в формате XML. Используется для проверки XML-документов. Определяет порядок следования и вложенности тегов, а также допустимые значения для их атрибутов.

EJB (Enterprise JavaBeans)
Серверная компонентная архитектура промышленного уровня. Названа в честь технологии JavaBeans, но не имеет к ней прямого отношения. Это коммерческие сервисы и компоненты баз данных, которые поддерживают транзакции и механизм безопасности, реализованный в декларативном стиле.

enum
Ключевое слово языка Java, которое используется для объявления перечислений. Перечисление хранит список константных объектов и может использоваться в качестве альтернативы числовым константам, которые служат идентификаторами или метками.

extends
Ключевое слово, с помощью которого при объявлении указывается родительский класс. Дочерний класс имеет доступ ко всем публичным и защищенным полям и методам своего родителя (или, если они находятся в одном пакете, вообще ко всем «не частным» элементам). Если при объяв-

лении это слово опущено, родительским считается класс `java.lang.Object`.

final

Ключевое слово, которое можно применять к классам, методам и переменным. Во всех трех случаях оно имеет немного разное значение. Класс, для которого указан этот модификатор, нельзя наследовать. Примером такого использования может служить класс `java.lang.System`. Если применить ключевое слово `final` к переменной, получится константа (то есть, ее значение уже нельзя будет изменить).

finalize

Зарезервированное имя метода. Метод `finalize()` вызывается Java-машиной, когда объект больше не используется (например, когда на него больше никто не ссылается), но перед тем, как его память будет утилизирована системой. Он нужен для того, чтобы очистить список задач и освободить системные ресурсы, прежде чем объект попадет в сборщик мусора.

finally

Ключевое слово, которое обозначает один из блоков конструкции `try/catch/finally`. Блоки `catch` и `finally` обрабатывают исключения и проводят очистку ресурсов для кода из блока `try`. Блок `finally` не является обязательным; его можно указать после блока `try` (при этом между ними может находиться любое количество блоков `catch`). Код, который в нем хранится, всегда выполняется только один раз, независимо от того, произошла ли ошибка в блоке `try`.

float

Простой тип в языке Java. Предназначен для хранения 32-битных чисел (одинарной точности) с плавающей запятой, представленных в формате IEEE 754.

HotJava

Очень старый веб-браузер, написанный на языке Java. Поддерживает загрузку и выполнение апплетов.

HTTP (Hypertext Transfer Protocol)

Протокол, который используется в веб-браузерах и других клиентских приложениях для взаимодействия с веб-серверами. В простейшем случае использует команды GET и POST для загрузки и отправки данных.

IDE (Integrated Development Environment)

Инструмент с графическим пользовательским интерфейсом, который поддерживает редактирование исходного кода, компиляцию, запуск, отладку и развертывание разрабатываемых приложений. Примеры: NetBeans, Eclipse и т. д.

implements

Ключевое слово, с помощью которого указывается интерфейс (интерфейсов может быть несколько), реализуемый классом. Не является обязательным. В объявлении оно должно следовать за оператором `extends` (если таковой имеется). Это ключевое слово заставляет класс (или один из его родительских классов) реализовать все методы из каждого указанного интерфейса.

import

Позволяет подключать внешние классы, используя их сокращенные имена, или устраняет неоднозначность, вызванную массовым подключением классов с помощью другого оператора `import`. (В языке Java доступ к классу всегда можно получить по его полному имени, при условии, что его можно найти в одной из директорий, указанной в переменной среды CLASSPATH, и его файл доступен для чтения. Оператор `import` не предоставляет доступ к классу — он просто экономит время программисту и делает код более понятным.) Программа на языке Java может содержать любое количество операторов `import`. Они должны быть указаны вверху исходного кода, после оператора `package`, но перед первым объявлением класса или интерфейса.

instanceof
Оператор в языке Java, который возвращает `true`, если по левую сторону находится объект типа (класса или интерфейса), указанного справа. В противном случае возвращает `false`. Значение `false` возвращается и в том случае, если указанный объект равен `null`.

int
Простой тип в языке Java, который представляет в дополнительном коде 32-битное число со знаком

ISO 8859-1
8-битная кодировка символов, стандартизированная организацией ISO. Известна также под названием Latin-1. Содержит символы латинского алфавита, которые подходят для английского и большинства других западноевропейских языков.

JavaBeans
1. Компонентная архитектура для платформы Java. Способ создания совместимых между собой объектов, которыми легко управлять с помощью визуальных средств разработки.
2. Классы, написанные в соответствии со спецификацией архитектуры JavaBeans.

JavaScript
Язык, разработанный компанией Netscape на ранних этапах становления Интернета. Предназначен для создания динамических веб-страниц. С точки зрения программиста он никак не связан с языком Java, хотя имеет отдаленно похожий синтаксис.

JAXB (Java API for XML Binding)
API-интерфейс на платформе Java, который позволяет генерировать классы на основе документов формата XML DTD или Schema (и наоборот).

JAXP (Java API for XML Parsers)
API-интерфейс на платформе Java, который позволяет создавать расширяемые и кроссплатформенные реализации инструментов для обработки данных в формате XML и XSL, в том числе анализаторы и преобразователи.

JAX-RPC
API-интерфейс на платформе Java, который позволяет обмениваться с веб-сервисами данными в формате XML, используя вызовы удаленных процедур (RPC).

JDBC (Java Database Connectivity)
Стандартный API-интерфейс на платформе Java, предназначенный для взаимодействия с базами данных посредством структурированного языка запросов SQL.

JDOM
Программный интерфейс для доступа к содержимому XML-документов, созданный Джейсоном Хантером и Бреттом Маклафлином. Является более простым в использовании, чем стандартный интерфейс DOM API. Использует коллекции и общепринятые соглашения. Доступен по адресу **http://www.jdom.org/**.

JWSDP (Java Web Services Developer Pack)
Стандартный набор расширений, упакованных вместе в виде устанавливаемого файла. Содержит различные пакеты для работы с форматом XML и веб-сервисами, включая JAXB и JAX-RPC.

Latin-1
Еще одно название кодировки ISO 8859-1.

long
Простой тип данных в языке Java, который представляет в дополнительном коде 64-битное число со знаком.

MVC (Model-View-Controller)
Архитектура пользовательского интерфейса родом из языка Smalltalk. В MVC данные называются *моделью* (Model), а способ вывода этих данных — *представлением* (View). Взаимодействие между ними обеспечивается *контроллером* (Controller). Концепции, на которых основана эта архи-

тектура, активно используются в платформе Java.

NaN (not-a-number)

Специальное значение, которое могут принимать переменные типа `float` и `double`. Представляет неопределенный результат математической операции — например, деление ноля на ноль.

native

Ключевое слово, позволяющее использовать методы, реализованные не на Java, а на языках программирования, которые компилируются в машинный код. Такие методы могут предоставлять доступ к сети, оконному интерфейсу, файловой системе и другим ресурсам.

new

Унарный оператор, который создает новый объект или массив (или генерирует исключение типа `OutOfMemoryException`, если не хватает свободной памяти).

NIO

«Новый» пакет для ввода/вывода на платформе Java. С появлением платформы Java 1.4 считается одним из основных инструментов. Поддерживает асинхронные, прерываемые и масштабируемые операции ввода/вывода. Дает возможность обрабатывать данные при помощи выборок, не привязываясь к отдельным потокам выполнения.

null

Специальное значение, которое сигнализирует о том, что ссылка не указывает ни на один объект. Является значением по умолчанию для статических и обычных полей.

package

С помощью оператора `package` указывается пакет для класса. Код, являющийся частью определенного пакета, имеет доступ ко всем его классам (публичным и не публичным), а также ко всем методам и полям этих классов, которые не являются частными. Класс, входящий в состав пакета, можно использовать только в том случае, если он хранится в соответствующей директории в рамках иерархии CLASSPATH. Весь код в файле, в котором нет оператора `package`, автоматически становится частью стандартного безымянного пакета. Это помогает при разработке небольших прототипов, которые запускаются из командной строки, потому что такой код может быть выполнен в текущей директории.

Preferences API

API-интерфейс платформы Java, который позволяет сохранять небольшие объемы пользовательской или системной информации между запусками Java-машины. Похож на небольшую базу данных или реестр в операционной системе Windows.

printf

Подход к форматированию текста, пришедший из языка Си. Его синтаксис основан на встроенных идентификаторах, а параметры передаются в виде списка аргументов переменной длины.

private

Ключевое слово `private` определяет доступ к методам и полям класса. Частный член класса невидим извне и недоступен для дочерних классов.

protected

Ключевое слово, которое определяет доступ к методам и полям класса. Защищенное поле является видимым для класса, всех дочерних классов и внутри текущего пакета. Обратите внимание, что дочерний класс, который находится в другом пакете, имеет доступ только к своим защищенным полям (или к полям тех объектов, которые являются его производными); он не может работать с защищенными полями экземпляров своего родительского класса.

public

Ключевое слово, которое определяет доступ к классам, интерфейсам, а также к их

методам и полям. Публичные классы и интерфейсы видны отовсюду. Непубличные классы и интерфейсы видно только внутри своего пакета. Публичные методы и переменные доступны в любой точке программы, из которой виден их класс. Если для поля не указать ни одного модификатора (`private`, `protected` или `public`), оно будет видимым только в рамках пакета, которому принадлежит его класс.

RMI (Remote Method Invocation)

Стандартная система распределения объектов на платформе Java. Позволяет передавать ссылки на объекты, находящиеся на удаленном компьютере, и вызывать из них методы так, как будто они доступны локально.

SAX (Simple API for XML)

API-интерфейс для анализа XML-документов, основанный на событийной модели. Клиент получает события в ответ на обнаружение открывающих и закрывающих тегов, символьных данных и т. д.

SDK (Software Development Kit)

Пакет программного обеспечения, предоставляемый разработчикам компанией Oracle. Содержит интерпретатор языка Java, классы и инструменты для разработки: компилятор, отладчик, дизассемблер, приложение для просмотра апплетов, генераторы шаблонных файлов и документации. Часто этот пакет называют JDK.

SecurityManager (диспетчер безопасности)

Класс, предоставляемый платформой Java, который содержит методы для проверки прав доступа к определенным операциям в контексте текущей среды выполнения.

short

Простой тип данных в языке Java, который представляет в дополнительном коде 16-битное число со знаком.

static

Ключевое слово, которое можно указать при объявлении метода или поля внутри класса. В отличие от переменной экземпляра, статическая переменная принадлежит своему классу. Каждая копия объекта имеет свой набор переменных экземпляра, но статические переменные у них совпадают, вне зависимости от количества созданных копий. Доступ к статическим переменным можно получить как по имени класса, так и через объект. Переменные, которые не являются статическими, доступны только в экземпляре класса.

static import

Конструкция, похожая на обычный оператор `import`, но при этом позволяет подключать имена статических методов и переменных в контекст текущего класса. Предоставляет некое подобие глобальных функций и констант.

String

Класс, который используется в языке Java для представления последовательности символов. Содержит множество методов для управления строковыми объектами.

super

Ключевое слово, которое используется для обращения к переменным и методам родительского класса. По принципу своей работы похоже на специальную ссылку `this`, с помощью которой можно сослаться на текущий контекст объекта.

synchronized

Ключевое слово в языке Java, которое используется двумя взаимосвязанными способами. Во-первых, оно может играть роль модификатора для методов (обычных и статических). Указывает на то, что изменения, которые метод вносит во внутреннее состояние класса или объекта, угрожают потоковой безопасности. Перед запуском статического метода, помеченного этим ключевым словом, Java-машина блокирует состояние класса, чтобы его

нельзя было изменить сразу из нескольких потоков выполнения. Перед запуском метода экземпляра, помеченного ключевым словом `synchronized`, Java-машина блокирует объект, который осуществил вызов, чтобы никакой другой поток не смог его изменить. Во-вторых, язык Java поддерживает выражение `synchronized`, с помощью которого обозначаются «важные» участки кода. Оно похоже на объявление метода: сначала идут круглые скобки, а потом одна или несколько конструкций. Выражению `synchronized` можно передавать объект или массив, которые будут заблокированы на время выполнения указанного блока кода.

TCP (Transmission Control Protocol)

Надежный протокол, ориентированный на установление соединений. Входит в семейство протоколов, на основе которых функционирует Интернет.

this

Находится внутри метода экземпляра или конструктора класса и указывает на «текущий объект». Позволяет ссылаться на поля, которые были перекрыты локальными переменными или аргументами метода. Также помогает передать текущий объект в качестве аргумента для статического метода (или метода другого класса). Если указать это ключевое слово в самом начале конструктора, оно автоматически вызовет один из других конструкторов текущего класса.

throw

Ключевое слово `throw` сигнализирует о том, что было выполнено условие, при котором должен быть сгенерирован определенный объект типа `Throwable` (исключение). При этом работа программы останавливается, а текущий поток выполнения переход к ближайшему блоку `catch`, способному обработать возникшее исключение.

throws

Ключевое слово `throws` используется для объявления списка исключений, которые могут быть сгенерированы методом. Исключения, которые не являются производными классов `Error` или `RuntimeException`, должны быть либо перехвачены внутри метода, либо объявлены с помощью выражения `throws`.

try

Ключевое слово `try` обозначает начало защищенного блока кода, к которому последовательно применяются блоки `catch` и `finally`. Само по себе не выполняет никаких особых действий. Больше информации о конструкции `try/catch/finally` ищите в разделах `catch` и `finally`.

UDP (User Datagram Protocol)

Протокол, основанный на датаграммах; работает без соединений, не обеспечивает надежную передачу данных и не поддерживает строгий контроль за пакетами.

Unicode

Универсальный стандарт для кодирования текста, разработанный одноименным консорциумом и вобравший в себя письменные знаки почти всех языков мира. В языке Java используется в типах `char` и `String`.

UTF-8 (8-битный формат преобразования кодировки Unicode)

Кодировка для символов в стандарте Unicode (и в целом для символов из набора, предусмотренного стандартом UCS), которая часто используется для передачи и хранения текстовой информации. Это многобайтовый формат, в котором разные символы могут быть представлены разным количеством байтов.

WAR (Web Applications Resources)

Разновидность формата JAR, которая имеет дополнительные структурные особенности и предназначена для хранения классов и ресурсов веб-приложений. WAR-файлы содержат директорию **WEB-INF**, в которой находятся классы и библиотеки, а также файл **web.xml** с инструкциями для развертывания приложения.

XInclude
Стандарт, основанный на формате XML и API-интерфейсах языка Java, который позволяет вставлять в XML-документы различные ресурсы.

XML (Extensible Markup Language)
Универсальный язык разметки текста и других данных. Использует вложенные *теги* для структурирования содержимого и добавления метаинформации.

XML Schema
Замена языка DTD, созданная консорциумом W3C. Язык, основанный на формате XML, который накладывает ограничения на структуру тегов, их атрибуты и допустимые типы данных в XML-документе. Имеет множество разновидностей с разным синтаксисом.

XPath
Стандарт, основанный на формате XML и API-интерфейсах языка Java, который предназначен для сопоставления элементов и атрибутов в XML-документах. Использует иерархию инструкций, похожих на регулярные выражения.

XSL/XSLT (Extensible Stylesheet Language/XSLTransformations)
Разновидность языка XML, предназначенная для описания инструкций по стилизации и преобразованию XML-документов. Стилизация подразумевает добавление разметки, которая обычно используется для вывода информации. XSLT — расширение языка XSL, которое позволяет полностью изменять структуру документов.

Агрегирование
Объединение имеющихся компонентов для создания нового, более сложного объекта. При этом разные задачи делегируются внутренним компонентам. Агрегирование отличается от наследования, при котором новый объект создается путем изменения или корректировки поведения старого объекта. См. *Наследование*.

Аннотации
Метаданные, которые добавляются в исходный код при помощи тега `@`. Могут использоваться на этапе компиляции или во время выполнения программы. Передают аргументы в классы, предоставляют данные, выполняют привязку или сигнализируют о дополнительных возможностях.

Апплет
Встраиваемое приложение на языке Java, которое работает в контексте программы просмотра апплетов, такой как веб-браузер.

Атомарный
Дискретный или транзакционный — в том смысле, что операция выполняется как единое целое (все или ничего). Атомарными являются некоторые операции, предоставляемые Java-машиной (JVM) и пакетом `java.util.concurrent`.

Буджумы (Boojum)
Мистическая, призрачная разновидность снарков. Из поэмы Льюиса Кэрролла «Охота на Снарка», написанной в 1876 году.

Веб-приложение
Программа, которая выполняется в рамках веб-сервера или сервера приложений. В качестве клиента обычно используется веб-браузер.

Веб-сервис
Сервис, который выполняется на стороне сервера и служит для взаимодействия с другими приложениями. В качестве формата данных выступает XML, а транспортным механизмом служит протокол HTTP.

Вектор
Динамический массив элементов.

Верификатор байт-кода
Механизм, который анализирует байт-код перед его запуском. Следит за тем, чтобы приложение нормально выполнялось и не конфликтовало с моделью безопасности

Java-машины. Это базовое средство защиты для платформы Java.

Внутренний класс
Класс, объявление которого содержится внутри метода или другого класса. Выполняется в рамках лексического пространства имен того класса, в котором находится.

Вызов типов
См. *Реализация типов*. Термин «вызов типов» иногда используется по аналогии с синтаксисом для вызова методов.

Глубокая копия
Дубликат объекта со всеми компонентами, на которые он ссылается. Глубокая копия дублирует всю иерархию объектов, а не только ссылки. См. также *Поверхностная копия*.

Границы
Диапазон, в котором может находиться обобщенный тип данных. Верхняя граница указывает на то, что тип должен наследовать определенный класс (или быть совместимым с ним). Нижняя граница говорит о том, что тип должен служить родительским классом для другого указанного типа (или быть совместимым с ним).

Графически пользовательский интерфейс (GUI)
Традиционный визуальный пользовательский интерфейс, состоящий из окон, внутри которых размещены разные графические элементы: кнопки, поля ввода, раскрывающиеся меню, диалоговые окна и т. д.

Графический контекст
Поверхность для рисования, представленная классом `java.awt.Graphics`. Содержит контекстную информацию об области вывода графики и предоставляет методы для выполнения операций рисования на ней.

Датаграмма
Пакет данных, обычно передающийся с помощью сетевых протоколов вроде UDP, которые не предоставляют гарантий относительно доставки, обработки ошибок или контроля информации.

Диспетчер компоновки
Объект, который в библиотеках Swing и AWT управляет размещением компонентов в рамках видимой области контейнера.

Дополнение (плагин)
Модульный программный компонент, предназначенный для расширения возможностей веб-браузера и обработки специфических типов данных (которые определяются по MIME-типу). Дополнение, поставляемое вместе с платформой Java, добавляет в веб-браузеры поддержку апплетов.

Дочерний класс
Наследник другого класса. От своего родителя получает публичные и защищенные методы и переменные. См. также *extends*.

Единица компиляции
Фрагмент исходного кода, содержащий определение класса (обычно только одного). В большинстве современных сред разработки единица компиляции эквивалентна файлу с расширением **.java**.

Загрузчик классов
Объект типа `java.lang.ClassLoader`, который выполняет загрузку двоичных классов в Java-машине. Помогает разграничивать классы в зависимости от их источника (для структурирования приложений и обеспечения безопасности). Может работать с иерархиями родительских/дочерних классов.

Имя сервера
Имя, которое выдается отдельному компьютеру, подключенному к сети. Имеет понятную для человека форму.

Инкапсуляция
Одна из методик объектно-ориентированного программирования, призванная ограничить доступ к переменным и методам,

а также упростить API-интерфейс классов и пакетов. Используя ключевые слова `private` и `protected`, программист может скрыть внутреннюю структуру класса (принцип «черного ящика»). Инкапсуляция снижает количество потенциальных ошибок, поощряя создание модульных компонентов, пригодных к многократному использованию. Этот подход еще называют *скрытием данных*.

Интерпретатор

Модуль, который декодирует и выполняет байт-код. Строго говоря, на платформе Java большая часть байт-кода давно не интерпретируется, а динамически компилируется в машинный код самой Java-машиной.

Интерфейс, interface

1. Ключевое слово, которое используется при объявлении интерфейсов.
2. Набор абстрактных методов, которые в совокупности определяют тип данных в языке Java. Классы могут реализовывать интерфейсы, в результате чего их экземпляры будут иметь тип этих интерфейсов.

Интроспекция

Механизм, с помощью которого интерфейсы технологии JavaBean предоставляют информацию о себе, дополняя данные, полученные в результате отражения.

Исключение

Сигнал о том, что с программой случилось нечто непредвиденное. В языке Java исключения являются объектами, наследованными от классов `Exception` или `Error` (которые сами являются производными класса `Throwable`). Для генерирования и обработки исключений предусмотрены ключевые слова `throw` и `catch` соответственно. См. также *catch*, *throw*, и *throws*.

Класс, class

1. Фундаментальная сущность, с помощью которой в большинстве объектно-ориентированных языков определяются объекты. Класс инкапсулирует набор переменных и методов, которые имеют привилегированный доступ друг к другу. Обычно объект является экземпляром класса с собственным уникальным набором данных.
2. Ключевое слово `class` используется для объявления новых классов (то есть, новых типов объектов).

Клиент

Потребитель ресурса или, в случае клиент-серверной архитектуры, сторона, которая инициирует взаимодействие. См. также *Сервер*.

Компилятор

Программа, которая транслирует исходный код в исполняемый.

Компонентная архитектура

Методика построения программ из отдельных частей. Автономные компоненты, пригодные для многократного использования, легко собираются в единое приложение.

Конструктор

Специальный метод, который автоматически вызывается при создании нового экземпляра класса. Используется для инициализации переменных нового объекта. Называется так же, как и сам класс, и не содержит в объявлении типа возвращаемого значения.

Контекст сервлета

Является частью технологии Servlet API и предоставляет сервлету доступ к ресурсам веб-приложения и сервера. Контекстом сервлета также часто называют базовый URL-адрес веб-приложения.

Криптография на основе открытого ключа

Криптографическая система, которая требует наличия закрытого и открытого ключей. Закрытый ключ позволяет расшифровывать сообщения, зашифрованные с помощью соответствующего открытого ключа (и наоборот). Открытый ключ можно сделать публично доступным без

нанесения ущерба безопасности; он также используется для проверки подлинности сообщений, отправленных обладателем закрытого ключа.

Легковесный компонент
Графический элемент управления, полностью написанный на языке Java и не связанный с аналогичным системным компонентом.

Локализация (интернационализация)
Процесс адаптации приложений для использования в разных странах, где разговаривают на разных языках. Иногда обозначается как I18N.

Локальная переменная
Переменная, объявленная внутри метода. Доступ к локальной переменной имеет только тот код, который находится в одном с ней методе.

Метод
Аналог функции или процедуры в объектно-ориентированном программировании.

Метод класса
См. *Статический метод*.

Метод экземпляра
Нестатический метод класса. Таким методам неявно передается ссылка `this`, указывающая на текущий объект. См. также *static*; *Статический метод*.

Модификатор
Ключевое слово, которое указывают перед классом, переменной или методом. Регулирует доступ к элементу, его поведение или семантику. См. также *abstract*; *final*; *native*; *private*; *protected*; *public*; *static*; *synchronized*.

Наследование
Важный аспект объектно-ориентированного программирования. Позволяет создавать новый объект на основе уже существующего, изменяя или дополняя его возможности. Новый объект по умолчанию содержит все переменные и методы своего родительского класса, которые не являются частными. Язык Java поддерживает одиночное наследование для классов, и множественное для интерфейсов.

Несформированный тип данных
Обобщенный класс без параметра, в котором указана информация о типе. После компиляции к этому типу приводятся все классы. См. также *Стирание типов*.

Обобщенные типы
Синтаксис и реализация параметризованных типов в языке Java, начиная с версии 5.0. Это классы, поведение которых можно изменять при помощи специальных параметров, хранящих дополнительные типы. В некоторых других языках их называют *шаблонами*.

Обобщенный класс
Класс, который использует синтаксис обобщенного типа и имеет один или несколько параметров для указания конкретных типов. Такие классы особенно хорошо подходят для создания контейнеров и коллекций с поддержкой разных типов элементов.

Обобщенный метод
Метод, который использует синтаксис обобщенного типа и имеет один или несколько параметров для задания типов аргументов и возвращаемого значения, которые будут использоваться во время выполнения программы. Часто компилятор может автоматически определять типы переменных по контексту, в котором используется метод.

Обработчик протокола
Компонент типа `URL`, который реализует сетевое соединение, необходимое для доступа к ресурсу по соответствующим протоколам (таким как HTTP или FTP). На платформе Java для обработки протокола используются два класса: `StreamHandler` и `URLConnection`.

Обработчик содержимого
Класс, который вызывается для разбора данных определенного типа и преобразования их в подходящий объект.

Объект
1. Фундаментальный структурный элемент объектно-ориентированного языка, который инкапсулирует данные и операции для управления этими данными.
2. Сущность, которая имеет структуру класса, но содержит собственную копию элементов данных. См. также *Экземпляр*.

Отражение
Возможность взаимодействовать со структурами языка программирования на этапе выполнения программы. В языке Java отражение позволяет анализировать class-файлы, искать внутри них члены класса, динамически вызывать методы и редактировать переменные.

Очередь
Структура данных, похожая на список, которая буферизирует рабочие элементы по принципу «первым пришел — первым ушел».

Параметризованный тип
Класс, который использует синтаксис обобщенных классов и зависит от одного или нескольких типов, указанных пользователем. Таким образом, класс адаптируется к работе с заданными типами.

Перегрузка методов
Позволяет объявлять сразу несколько методов с одним и тем же именем, но с разным списком аргументов. При вызове перегруженного метода компилятор проверяет типы аргументов, выбирая подходящую версию.

Перекрытие переменных
Ситуация, когда в родительском и дочернем классах есть переменная с одним и тем же именем. При этом переменная дочернего класса «перекрывает» родительскую версию. К перекрытым переменным можно обращаться с помощью ключевого слова `super` или путем приведения текущего объекта к типу родительского класса.

Переменная класса
См. *Статическая переменная*.

Переменная экземпляра
Нестатическая переменная класса. Каждый объект содержит свою отдельную копию всех переменных экземпляра своего класса. См. также *Переменная класса*; *static*.

Переопределение методов
Объявление метода с точно такими же именем и типами аргументов, как у метода из родительского класса. При вызове переопределенного метода интерпретатор динамически определяет, какая из версий подходит для текущего объекта. С выходом платформы Java 5.0 переопределенные методы могут иметь разные типы возвращаемых значений (с некоторыми оговорками).

Перечисление
См. *enum*.

Поверхностная копия
Копия объекта, которая дублирует только содержащиеся в нем значения. Ссылки на другие объекты не копируются. См. также *Глубокая копия*.

Подписанный апплет
Апплет, упакованный в JAR-файл с цифровой подписью, с помощью которой можно проверить его происхождение и целостность его содержимого.

Подписанный класс
Класс (или архив) с цифровой подписью, которая позволяет проверить его происхождение и исключить возможность изменения содержимого. Это позволяет выдавать права на выполнение класса, не подвергаясь никакому риску.

Подстановочный тип
Синтаксис языка Java, который позволяет подставлять в обобщенные классы звездочку «*» вместо настоящего типа. Сигнализирует о том, что обобщенный класс является набором из множества отдельных типов.

Поле ввода со счетчиком
Элемент пользовательского интерфейса, отображающий значение, которое можно увеличивать и уменьшать с помощью двух небольших кнопок, размещенных справа. Компонент `JSpinner` из библиотеки Swing поддерживает диапазоны чисел и дат, а также может работать с произвольными значениями.

Полиморфизм
Один из фундаментальных принципов объектно-ориентированного программирования, который гласит, что два типа, связанные наследованием, совместимы между собой. То есть наследник может использоваться вместо своего родителя, расширяя и изменяя его возможности.

Поток ввода/вывода
Поток данных или коммуникационный канал. Все основные операции ввода/вывода на платформе Java основаны на потоках. В технологии NIO используются каналы, которые ориентированы на пакетную передачу данных.

Поток выполнения
Автономный контекст выполнения кода внутри программы. Так как Java является многопоточным языком программирования, JVM может выполнять одновременно несколько потоков. На платформе Java потоки представлены объектами типа `Thread`.

Приведение типов
Ручное назначение объекту определенного типа. Приведение типов в языке Java проверяется как статически, с помощью компилятора, так и во время выполнения программы.

Приложение
Программа на языке Java, которая в отличие от апплетов работает автономно.

Простой тип
Один из встроенных типов языка Java: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` или `double`. Простые типы управляются, присваиваются и передаются в методы «по значению» (то есть, происходит копирование данных). См. также *Ссылочный тип*.

Пул потоков
Группа потоков, предназначенных для многократного выполнения служебных запросов. Поток изымается из пула, выполняет задачу и возвращается обратно.

Распаковка
Извлечение простых значений из объектов-оберток.

Реализация типов
Этап, на котором в обобщенный класс передаются параметры с реальными типами. В результате из обобщенного класса создается новый тип.

Регулярное выражение
Лаконичный, но мощный синтаксис для описания шаблонов в тексте. Может использоваться для распознавания и разбора большинства текстовых конструкций, позволяя охватывать широкий спектр их вариаций.

Родительский класс
Класс, который наследуется другим классом. Передает своему потомку публичные и защищенные методы и переменные. См. также *Extends*.

Сборка мусора
Процесс освобождения памяти, принадлежащей объектам, которые больше не используются. Таковыми считаются объекты, на которые никто не ссылается (ни другие объекты, ни локальные переменные, чей метод находится в стеке вызовов).

Сервер
Система, которая предоставляет ресурс или, в контексте клиент-серверной архитектуры, принимает запрос для дальнейшего взаимодействия. См. также *Клиент*.

Сервлет
Программный компонент, который является реализацией интерфейса `javax.servlet.Servlet`. Может работать внутри специального контейнера или веб-сервера. Широко используется в веб-приложениях для обработки данных и генерирования разных видов информации, включая HTML-страницы.

Сериализация
«Сериализовать» означает упорядочить, разместить последовательно. При сериализации объект упаковывается таким образом, чтобы его можно было хранить или передавать по сети. Методы такого объекта синхронизируются относительно текущего потока выполнения, чтобы их можно было вызывать исключительно последовательно.

Сертификат
Электронный документ с цифровой подписью, который используется для идентификации человека, группы или организации. Сертификат удостоверяет конкретную личность или группу лиц и должен содержать открытый ключ. Подписываются в центрах сертификации с помощью цифровой подписи.

Сигнатура
1. Цифровая подпись. Хеш-код сообщения, зашифрованный с помощью скрытого ключа и сертификата, которые принадлежат подписывающей стороне. Адресат может получить из сертификата открытый ключ, расшифровать хеш-код и сравнить его с хеш-кодом подписанного сообщения. Если значения совпадают, адресат может быть уверен в том, что сообщение не изменялось и что владелец подписи является подлинным.
2. Название, аргументы и иногда тип возвращаемого значения, которые в совокупности однозначно идентифицируют метод, находящийся в определенном контексте.

Скрытие данных
См. *Инкапсуляция*.

Событие
1. Действие пользователя, такое как щелчок мышью или нажатие клавиши.
2. Объект, доставляемый зарегистрированному слушателю событий в ответ на действие пользователя или какую-то системную активность.

Сокеты
Сетевой API-интерфейс, впервые появившийся в операционной системе BSD Unix. Два сокета представляют собой конечные точки взаимодействия между двумя узлами сети. Серверный сокет отслеживает подключения, инициируемые клиентами, создавая отдельные сокеты для каждой сессии.

Список аргументов переменной длины
В языке Java методы могут принимать любое количество аргументов заданного типа; при этом они должны быть указаны после четко определенного списка параметров. Такие аргументы автоматически упаковываются в массив.

Ссылочный тип
Любой объект или массив. Ссылочные типы управляются, присваиваются и передаются в методы «по ссылке». Другими словами, копируется не исходное значение, а ссылка на него. См. также *Простой тип*.

Статическая переменная
Переменная, помеченная ключевым словом `static`. Такие переменные связаны непосредственно со своим классом, а не с каким-либо его экземпляром. Вне зависимости от количества объектов, созданных на основе класса, статические переменные

всегда существуют в единственном экземпляре.

Статический метод

Метод, помеченный ключевым словом `static`. Такие методы не передают ссылку `this` и могут работать только со статическими переменными и методами текущего класса. Доступ к ним можно получить только через имя класса, но не через его экземпляр.

Стирание типов

Методика, применяемая в языке Java для обобщенных типов. Суть ее в том, что во время компиляции тип объекта удаляется (стирается), а вместо него используется универсальный тип `Object`. При этом обеспечивается обратная совместимость с кодом, который не поддерживает обобщенные типы. Приводит к некоторому усложнению языка.

Тег <param>

Тег языка HTML, который используется внутри другого тега, `<applet>`. С его помощью апплету можно передавать параметры в виде пар «ключ/значение».

Упаковка

Помещение простых типов данных внутрь соответствующих оберток. См. также *Распаковка*.

Функция обратного вызова

Блок кода, который определен в одном объекте, а вызывается из другого. Вызов происходит при наступлении определенного события. Механизм событий на платформе Java основан на функциях обратного вызова.

Хеш

Число, полученное криптографическим методом; используется для определения того, вносились ли в содержимое сообщения какие-либо изменения. Вместе с сообщением изменяется и его хеш. Надлежащая реализация алгоритма хеширования практически исключает возможность создания двух разных сообщений с одним хешем.

Хеш-код

Идентификатор, который выглядит как случайное число. Основан на содержимом объекта и используется в качестве его подписи. С помощью хеш-кодов объекты помещаются в хеш-таблицы. См. также *Хеш-таблица*.

Хеш-таблица

Объект, похожий на словарь или ассоциативный массив. Позволяет хранить и извлекать элементы с помощью ключей, которые называются «хеш-коды». См. также *Хеш-код*.

Центры сертификации

Организация, которой поручено следить за подлинностью сертификатов. Для этого могут использоваться разные средства, вплоть до проверки документов, удостоверяющих личность владельца сертификата.

Цепочка исключений

Шаблон проектирования, заключающийся в перехвате одного исключения и генерировании вместо него другого, более высокоуровневого или подходящего. При этом внутри нового исключения находится и *исходное*, которое при необходимости можно извлечь.

Экземпляр

Наличие чего-либо (обычно объекта). Создавая объект из класса, мы получаем *экземпляр* этого класса.

ОБ АВТОРАХ

Патрик Нимейер познакомился с языком программирования Oak (предшественником Java) во времена своей работы в компании Southwestern Bell Technology Resources. В настоящее время он занимает должность технического директора в фирме Ikayzo, Inc., что не мешает ему быть независимым консультантом и писателем. Патрик является автором и участником множества проектов с открытым исходным кодом, среди которых можно выделить его детище — популярный скриптовый язык программирования для платформы Java под названием BeanShell. Он был членом несколько экспертных советов, которые формировали список возможностей для новых версий языка Java в рамках инициативы JCP. С недавних пор Патрик занялся разработкой аналитического программного обеспечения для финансовой индустрии, а также созданием сложных мобильных приложений. Сейчас он вместе со своей семьей и разными домашними питомцами проживает в городе Сент-Луисе.

Дэн Льюк занимает должность генерального директора фирмы Ikayzo, Inc., которая базируется сразу в двух городах, Токио и Гонолулу, и занимается разработкой программного обеспечения и интерактивных веб-сайтов; среди клиентов этой фирмы можно выделить такие компании как Sony, Oracle, Nomura и PIMCO, а также федеральное правительство США. Дэн имеет колоссальный опыт работы; в его послужном списке есть должности старшего вице-президента по исследованиям и разработке (токийская компания ValueCommerce, которая является крупнейшим производителем интернет-рекламы во всей Азии), директора по международному развитию (лондонская фирма LastMinute.com, крупнейший в Европе вебсайт по предоставлению различных услуг) и президента (американское подразделение компании DML). Он давно и успешно руководит группами программистов из более чем полутораста человек в пяти разных странах. На его счету участие в многочисленных консультативных советах, проводимых такими компаниями, как Macromedia и Sun Microsystems. Дэн является активным членом сообщества Java-программистов; он помогает разрабатывать язык BeanShell, руководит проектом SDL и регулярно посещает заседания, проводимые в рамках инициативы JCP.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Apache Tomcat, 727

big-endian, 627

FIFO, 522

HTML5, 728, 732, 795

Internet Protocol, 645
IP, 645

JAF, 712
JAI, 713
Java Activation Framework, 712
Java Advanced Imaging API, 713
Java Media Framework, 713
Java-апплеты, 545
JMF, 713
JNDI, 682
JSP-сценарии, 727, 729, 730, 757

little-endian, 627

RPC, 722
Rласс JCheckBox, 848

Security Manager, 650
SOCKS, 662
SUID, 613

WAR-файлы, 732
WebLogic, 727

Zip Filesystem Provider, 619
ZIP-архив, 616, 598, 614

Абстрактно-оконный инструментарий (AWT), 52, 92, 93
Адаптер, 834
Адаптивная компиляция, 32
Адрес URL, 588, 590, 595, 596, 597, 619
Алгоритм rot13, 581
Алгоритм Нейгла, 660
Алгоритм хеширования, 530, 532
Аннотации, 149, 269, 314, 315, 319, 320, 321, 322
Аннотация @author, 149
Аннотация @deprecated, 149
Аннотация @exception, 148, 149
Аннотация @param, 149
Аннотация @return, 149
Аннотация @see, 148, 149
Аннотация ServletSecurity, 768
Аннотация @since, 149
Аннотация @TestValues, 320
Аннотация @version, 149
Аннотация WebFilter, 775
Аннотация WebServlet, 756
Анонимные внутренние классы, 280, 281, 288
Анонимные массивы, 203
Архитектура MVC, 800, 804, 888, 899
Ассоциативный массив, 524

Библиотека AWT, 797, 803, 815, 838
Библиотека JFC, 796
Библиотека Swing, 715, 796
Битовые поля, 539
Блок synchronized, 537
Блоки кода, 30, 44, 160, 167, 253, 388
Блокирующая очередь, 523
Брандмауэр, 661
Буфер, 574, 622

Ввод/вывод
выборочный, 621
неблокирующий, 621
Веб-приложения, 705
Веб-сервисы, 705, 722
Веб-сокеты, 732
Ветки case, 163
Виртуальная сетка, 952, 953, 962
Вспомогательное хранилище, 548
Выборочный ввод/вывод, 694
Выход за пределы допустимого диапазона, 497

Гауссово распределение, 502
Генератор псевдослучайных чисел, 501

Датаграммы, 645, 664
Двоичное дерево поиска, 529
Двунаправленная очередь, 523
Декодер, 571, 628, 630
Декоративное обрамление, 857
Демонический поток, 381
Деревья, 529
Десериализация, 611, 612
Диалоговые окна, 880
Диспетчер безопасности, 650, 658
Диспетчеры компоновки, 799, 810, 940
Древовидные списки, 908

Журнал сообщений, 550

Задержка, 513
Значение false, 121, 152
Значение NaN, 497

Идентификатор %g, 557
Идентификатор %tc, 510
Идентификатор URI, 599, 620
Идентификатор URN, 720
Идентичность, 533
Издатель, 560
имя сервера, 645
Интерактивное телевиденье (ITV), 28
Интерпретатор, 30, 31, 32
Интерфейс ActionListener, 110, 801, 821, 824, 835

Интерфейс Autocloseable, 190
Интерфейс BlockingQueue, 419, 523
Интерфейс Border, 857
Интерфейс ByteChannel, 623
Интерфейс Callable и класс Future, 410
Интерфейс Cloneable, 298
Интерфейс Collection, 515
Интерфейс Comparable, 359, 369
Интерфейс ConcurrentMap, 526
Интерфейс Deque, 523
Интерфейс Document, 889
Интерфейс FileVisitor, 607
Интерфейс Filter, 774, 776
Интерфейс FocusListener, 829
Интерфейс Iterator, 519
Интерфейс java.rmi. Remote, 679
Интерфейс java.util. Enumeration, 520
Интерфейс java.util.Map, 524
Интерфейс List, 521
Интерфейс ListSelectionModel, 853
Интерфейс MouseInfo, 838
Интерфейс MouseListener, 822
Интерфейс MouseMotionListener, 822
Интерфейс MouseWheelListener, 829
Интерфейс MutableTreeNode, 909
Интерфейс NavigableMap, 526
Интерфейс NavigableSet, 521
Интерфейс Queue, 522
Интерфейс Scheduled ExecutorService, 754
Интерфейс Serializable, 609, 610
Интерфейс ServletContext, 752, 753
Интерфейс Set, 521
Интерфейс SortedSet, 521
Интерфейс SwingConstants, 846
Интерфейс TableModel, 917
Интерфейс TreeNode, 909
Интерфейс WatchService, 608
Интроспекция, 825
Исключение ArithmeticException, 496
Исключение ArrayStoreException, 280, 338
Исключение BackingStoreException, 548
Исключение BadExpression, 923
Исключение CancellationException, 413
Исключение ClassCastException, 518
Исключение Closed ChannelException, 634
Исключение IllegalStateException, 520, 524
Исключение Malformed URLException, 707
Исключение ParseException, 509
Исключение RemoteException, 679
Исключение ServletException, 736
Исключение TooMany ListenersException., 825
Исключение Unknown ServiceException, 708
Исключение Unsupported OperationEx, 538
Исключение Unsupported OperationException, 520
Исключения, 122, 123, 124
Итератор, 516, 519, 526

Каналы, 622
Класс AccessibleObject, 308
Класс ActionEvent, 807
класс Animation, 373, 374, 375, 376
Класс Array, 197, 313
Класс ArrayList, 517, 532
Класс AWTKeyStroke, 908
Класс Aysnchronous FileChannel, 638
Класс BigDecimal, 499
Класс BigInteger, 499
Класс BorderFactory, 858
Класс BorderLayout, 811, 945, 948
Класс Box, 948
Класс BoxLayout, 893, 948
Класс BufferedInputStream, 572
Класс BufferedReader, 648
Класс Button, 85, 86
Класс ButtonGroup, 848
Класс ByteArrayInputStream, 580
Класс ByteArrayOutput Stream, 580
Класс ByteBuffer, 624, 626
Класс ByteOrder, 628
Класс CardLayout, 941, 950
Класс CharBuffer, 624, 627
Класс Charset, 629
Класс CharsetDecoder, 630
Класс CharsetEncoder, 630
класс Class, 301, 302, 304, 311, 345
Класс Color, 113, 114
Класс ComponentEvent, 826
Класс Concurrent LinkedQueue, 537
Класс Container, 800
Класс CopyOnWriteArrayList, 538
Класс CopyOnWriteArraySet, 538
Класс DatagramSocket, 645, 664
Класс DataInputStream, 573
Класс DataOutputStream, 573
Класс Date, 651
Класс DateFormat, 508
Класс Desktop, 818
Класс DirectoryStream, 600, 605
Класс EventObject, 820
Класс Exception, 185
Класс FileChannel, 631
Класс FileInputStream, 590
Класс FileLock, 634
Класс FileOutputStream, 590
Класс Files, 599
Класс FileSystem, 599
Класс FileVisitor, 600
Класс FilterInputStream, 573
Класс FilterOutputStream, 573
Класс FilterReader, 573
Класс FilterWriter, 573
Класс FlowLayout, 811, 942
Класс GridBagConstraints, 953
Класс GridBagLayout, 811, 952
Класс GridLayout, 811, 900, 944
Класс GZIPOutputStream, 614
Класс HashMap, 524

Класс HashSet, 532
Класс HttpServlet, 733, 734
Класс HttpSession, 744
Класс HttpURLConnection, 708, 710, 719
Класс IdentityHashMap, 533
Класс ImageIcon, 844
Класс InetAddress, 668, 670
Класс InputEvent, 827
Класс InputStream, 566
Класс InputStreamReader, 567
Класс InputVerifier, 896, 897
Класс Insets, 965
Класс JApplet, 810
Класс java.awt.Robot, 838
Класс java.io.File, 583
Класс java.lang.Math, 496, 497
Класс java.text.DateFormat, 503
Класс java.util.Calendar, 502
Класс java.util.concurrent.ConcurrentHashMap, 537
Класс java.util.Date, 502
Класс java.util.GregorianCalendar, 502
Класс java.util.Hashtable, 514
Класс java.util.Properties, 542
Класс java.util.Random, 501
Класс java.util.Vector, 514
Класс JButton, 844
Класс JCheckBoxMenuItem, 865
Класс JColorChooser, 887
Класс JComboBox, 851
Класс JComponent, 800
Класс JEditorPane, 886, 900
Класс JFileChooser, 884
Класс JFormattedTextField, 892, 893
Класс JFrame, 810
Класс JLabel, 845
Класс JList, 851
Класс JMenu, 861
Класс JMenuBar, 861
Класс JMenuItem, 863
Класс JOptionPane, 881
Класс JPanel, 810
Класс JPasswordField, 897
Класс JPopupMenu, 865
Класс JRadioButton, 848
Класс JScrollPane, 854, 870
Класс JSlider, 878
Класс JSpinner, 854
Класс JSplitPane, 872
Класс JTabbedPane, 874, 950
Класс JTable, 914
Класс JTextArea, 888
Класс JTextComponent, 888, 891
Класс JTextField, 888
Класс JTextPane, 904
Класс JToolBar, 903
Класс JTree, 908
Класс KeyboardFocusManager, 808, 907
Класс KeyStroke, 864
Класс LayoutManager, 940
Класс LinkedList, 532, 533
Класс MouseEvent, 821
Класс Naming, 682
Класс ObjectInputStream, 610
Класс ObjectOutputStream, 610
Класс OutputStream, 566
Класс OutputStreamWriter, 567
Класс Path, 599
Класс PipedInputStream, 577
Класс PipedOutputStream, 577
Класс PopupMenu, 828
Класс PrintStream, 567
Класс PrintWriter, 575
Класс PriorityQueue, 533
Класс ProxySelector, 663
Класс RandomAccessFile, 593, 594
Класс Reader, 566
Класс RemoteException, 175
Класс ScrollPane, 829
Класс ServerSocket, 645
Класс Servlet, 733
Класс SingleThreadModel, 734
Класс Socket, 645
Класс SpinnerDateModel, 856
Класс SpinnerListModel, 855
Класс SpinnerNumberModel, 855
Класс Stack, 523
Класс StringReader, 579
Класс StringTokenizer, 923
Класс SwingWorker, 841
Класс SystemTray, 818
Класс TableRowSorter, 924
Класс Timer, 512
Класс TimerTasks, 512
Класс TimeZone, 505
Класс TreeMap, 532
Класс TreeSet, 532
Класс UIManager, 931
Класс UnicastRemoteObject., 680
Класс URL, 706
Класс URLConnection, 710
Класс URLDecoder, 719
Класс URLEncoder, 715
Класс WeakHashMap, 538
Класс Writer, 566
Класс ZipEntry, 616
Класс ZipOutputStream, 616
Классы-обертки, 496
Ключевое слово transient, 610, 673
Ключ/значение, 514, 515, 524, 525, 542, 543, 546
Кодек, 571, 630
Кодировка, 567
Кодировка Latin-1, 570
Кодировка Unicode, 564, 567, 570, 571, 574, 582, 628, 629
Кодировщик, 571, 628
Коллекции, 514
Кольцевой буфер, 553
Команды действия, 844
Комментарии блока, 147
Компаратор, 533
Компонент JavaBean, 612
Конвейер, 578
Конструкция try-with-resources, 569, 591, 593, 605, 620, 635
Контейнеры, 799, 810
Контекстное меню, 865
Контроллер, 804
Копирование при записи, 536
Кэш, 524

Легковесные компоненты, 802, 803
Локальные переменные, 216

Мантисса, 500
Массив, 113, 527
Меню эпизодов. См. Меню глав
Мера наивысшей точности, 501
Метод abs(), 499
Метод actionPerformed(), 110, 111
Метод addMouseMotionListener(), 99, 100, 103
Метод appendReplacement(), 492

Метод appendTail(), 492, 493
Метод aquire, 431
Метод arraycopy(), 201
Метод asList(), 519
Метод binarySearch(), 530
Метод Calendar.add(), 504
Метод Calendar.roll(), 504
Метод cancel(), 413
Метод clone(), 297, 298, 300, 307, 308
Метод cos(), 497
Метод DateFormat.getInstance(), 508
Метод DateFormat.parse(), 509
Метод Double.isNaN(), 497
Метод doubleToRawLongBits(), 501
Метод drawString(), 94
Метод Float.isNaN(), 497
Метод floatToIntBits(), 501
Метод getCause(), 185
Метод getClass(), 301, 307
Метод getConstructor(), 307, 313
Метод getDateInstance(), 508
Метод getDateTimeInstance(), 508
Метод getExponent(), 501
Метод getHours(), 172
Метод getTime(), 505
Метод getTimeInstance(), 508
Метод hashCode(), 535
Метод hashCode()., 530
Метод indexOf(), 447, 449
Метод log(), 499
Метод Logger.getLogger()., 551
Метод Math.nextUp(), 501
Метод Math.random(), 502
Метод Math.round(), 499
Метод Math.ulp(), 500
Метод max(), 499
Метод min(), 499
Метод mouseMoved(), 99, 100, 101, 102, 103, 104
Метод nextGaussian(), 502
Метод paint(), 806
Метод paint()., 805
Метод paintComponent(), 806
Метод pow(), 499
Метод repaint(), 806
Метод schedule(), 513
Метод service(), 734
Метод setBoolean(), 308
Метод setContentType(), 737
метод setInt(), 308, 309
Метод setLocale(), 508
Метод setTime(), 505
Метод setTimeZone(), 505
Метод shutdown(), 414, 415
Метод sin(), 497
Метод sqrt(), 497, 499
Метод System.currentTimeMillis(), 503
Метод System.getProperties(), 545
Метод TimeZone.getAvailableIDs(), 505
Методы-обертки, 518
Многострочные поля ввода, 888
Модальные окна, 881
Модель данных, 833, 851, 909
Модификатор abstract, 262
Модификаторы, 128, 210, 211, 268, 275, 277, 282, 288, 293
Модульная величина, 500
Мягкая ссылка, 238

Набор, 521
Набор ресурсов, 559
Надежный протокол, 644
Наследование, 39, 89, 95, 104, 207, 230, 237, 244, 336, 337
Ненадежный протокол, 644
Необработанные типы, 333, 354
Номер порта, 645

Обобщение, 55, 259, 333
Обобщенные типы, 514
Обработчики, 552
Общение. См. Обсуждение
Общие классы, 356, 357, 358
Общие методы, 357, 358, 361, 362, 363
Объект AsyncContext, 756
Объект Charset, 571
Объект CompletionHandler, 639
Объект EditorKit, 903
Объект FilterChain, 775
Объект Graphics, 806
Объект Icon, 844
Объект Insets, 811
Объект javax.servlet.http.Cookie, 750
Объект MappedByteBuffer, 636
Объект Principal, 773
Объект RequestDispatcher, 753
Объект SelectionKey, 695
Объект Selector, 695
Объект StringIterator, 689
Объекты, 85, 86
Объекты Proxy, 663
Объекты для передачи данных, 679
Объекты форматирования, 553
Однострочные поля ввода, 888
Оператор instanceof, 675, 851
Операторы break/continue, 166
Операционная система Android, 29
Опция SO_LINGER, 660
Опция SO_TIMEOUT, 659
Опция TCP_KEEPALIVE, 661
Опция TCP_NODELAY, 660
Отложенное связывание, 40, 41
Относительном позиционировании, 965
Относительный путь, 586
Отображение файлов в память, 635
Отражение, 825
Очередь, 522

Пакет AWT, 709
Пакет java.awt, 810
Пакет java.awt.event, 826
Пакет java.awt.image, 797
Пакет java.awt.peer, 803
Пакет java.net, 641, 643
Пакет java.nio, 641
Пакет java.nio.file, 583, 599
Пакет java.rmi, 641
Пакет java.util.logging, 550
Пакет java.util.zip, 614
Пакет javax.jws, 727
Пакет javax.servlet, 726
Пакет javax.servlet.http, 734
Пакет javax.swing, 796
Пакет javax.swing.border, 857
Пакет javax.swing.event, 826
Пакет NIO, 563, 588, 598, 619, 622, 639, 753
Панель инструментов, 903
Параметризация, 516
Первым пришел первым ушел, 522
Перегрузка, 227, 229
Переключатели, 848
Переключатель, 483

Переменная среды classpath, 595
Переопределение, 248
Перечисления, 539
Планировщик, 512
Подкласс, 110
Подписчик, 560
Поле File.separator, 585
Поле File.separatorChar, 585
Поле serialVersionUID, 613
Поле ввода со счетчиком, 855
Ползунковые регуляторы, 878
Политика безопасности, 47, 141
Полный путь, 586
Полосы прокрутки, 878
Последним пришел первым ушел, 522
Потоки ввода/вывода, 563
Предусловия, 195
Преобразование строк, 464
Приведение типов, 256, 258, 259, 298, 325, 332, 335, 339, 514
Проверенные исключения, 182, 183, 254
Проект Apache CXF, 723
Прокси-серверы, 662
Промежуточный буфер, 805
Протокол HTTP, 705
Протокол ICMP, 670
Протокол SMTP, 646
Протокол TCP, 645
Протокол UDP, 645
Прямые буферы, 622
Пул потоков, 623, 638
Путь к ресурсу, 596

Рабочая директория, 584
Равенство, 533
Разделитель, 585
Разделяемая блокировка, 634
Разделяемая панель, 872
Раскрывающиеся списки, 853
Регистраторы, 551
Регулярные выражения, 474
Реестр удаленных объектов, 681, 682

Связной список, 528
Связующие объекты, 803
Связывание динамических методов, 252
Селекторы, 638, 694
Семафор, 431, 432
Сервер приложений, 726
Сервлеты, 705
Сериализация, 609
Сессии, 743
Сетевой порядок байтов, 573
Сети. См. Социальные сети
Символ преобразования " , 466, 467
Синхронизированные коллекции, 535
Синхронизированные потоки, 392
Системные настройки, 547
Ситуативный полиморфизм, 228
Скрытые переменные, 251, 257
Слабые ссылки, 238, 533, 538
Словарь, 524
Слушатели событий, 549
Событие FormSubmitEvent, 903
Событие HyperlinkEvent, 900
Сокеты, 643
Специальный символ %h, 557
Списки, 851
Список, 521
Список с пропусками, 521, 529
Стандарт IEEE 754, 500
Стандартный ввод, 567
Статические члены, 112, 113
Статический метод, 218, 219
Стек компонентов, 950

Таймер, 513
Технология AJAX, 729, 730, 732
Технология CGI, 727
Технология Collections Framework, 514
Технология CORBA, 693
Технология JavaBeans, 712, 802
Технология JAXP, 723
Технология JAX-WS, 723
Технология Preferences API, 546
Технология RMI, 677
Технология RPC, 677
Технология Servlet API, 732, 743, 753, 759, 766, 767
Технология SSL, 770
Тип данных char, 146
Трассировка стэка, 180

Угловые скобки (<>), 326, 340
Удаленный вызов методов, 677
Удаленный вызов процедур, 677, 722
Удаленный интерфейс, 679
Уникальный идентификатор сериализованной, 613
Утилита Ant, 786
Утилита make, 38, 134, 271
Утилита rmiregistry, 682
Утилита serialver, 614
Утилита wsgen, 723
Утилита wsimport, 723

Файловые системы, 598
Файлы cookie, 744
Фильтрация сообщений, 553
Фильтрующие потоки, 572
Флажки, 847
Формат ASCII, 582
Формат GZIP, 614
Формат HTML, 705
Формат XML, 544
Формат ZIP, 614
Форматирующая строка, 462, 463, 466, 468, 469, 472, 474
Фрактал, 422, 425, 426
Фреймворк Fork/Join, 421, 422
Фреймворк GWT, 731
Фреймворк JSF, 731
Фреймворк Spring, 731
Фреймворк Struts, 730, 731

Хеш, 533
Хеш-таблица, 542
Хеш-таблицы, 530

Часовой пояс, 503, 505
Частота, 513
Чиповые карты, 29
Числа произвольной точности, 499

Шаблоны подстановки, 605

Эксклюзивная блокировка, 634
Эра Unix, 587, 653

Язык WSDL, 722
Язык XML Schema, 762, 793, 794
Язык XSL, 730

Программирование на **Java**

Если вы новичок, то перед вами практическое пособие по изучению свойств и интерфейса языка Java. Авторы Патрик Нимейер и Дэниэл Леук, используя великолепные, реалистичные и забавные примеры, познакомят вас с основами языка Java, включая его библиотеки классов, приемы программирования и идиомы, а также примеры создания реальных приложений.

Если у вас уже есть опыт работы с Java, эта книга быстро выведет вас на требуемый уровень по параллельным программам, таким как приложения Java для веб-сервисов и XML. Вы изучите мощные новые средства управления ресурсами и исключениями в ваших приложениях.

- **Изучайте базовый язык Java – базовый синтаксис, классы и объекты и другие компоненты**
- **Разрабатывайте на Java, используя компилятор, интерпретатор и другие инструменты**
- **Исследуйте встроенные средства для работы с потоками и пакеты для поддержки параллелизма**
- **Осваивайте обработку текста и интерфейс мощных регулярных выражений**
- **Пишите современные сетевые или базирующиеся на веб-технологиях приложения и сервисы**
- **Создавайте графические приложения, используя компонентную архитектуру Java**
- **Осваивайте новые мощные функциональные интерфейсы, в том числе и только что появившиеся в Java 8**
- **Работайте с XML и XSLT, включая XML Schema, валидацию, XPath и XInclude**

Об авторах:

Патрик Нимейер – независимый консультант и автор в области сетевых технологий и распределенных приложений, начавший изучать язык Oak (предшественник Java) в корпорации Southwestern Bell Technology Resources. Он создал скриптовый язык BeanShell, основанный на Java.

Дэн Леук – директор корпорации Ikayzo с офисами в Токио и Гонолулу, которая занимается интерактивным дизайном и разработкой программного обеспечения. В число клиентов Ikayzo входят Sony, Oracle, Nomura, PIMCO и правительство США. Дэн Леук активно участвует в деятельности Java-сообщества и является председателем нескольких экспертных групп Java Community Process.

В этой книге есть все. Авторы проделали огромную работу.
Я планирую пользоваться этой книгой буквально каждый день.

Кристофер Бейсингер, программист

ISBN 978-5-699-71504-6

9 785699 715046 >

ЭКСМО

O'REILLY®

Об авторах:

Патрик Нимейер – независимый консультант и автор в области сетевых технологий и распределенных приложений, начавший изучать язык Oak (предшественник Java) в корпорации Southwestern Bell Technology Resources. Он создал скриптовый язык BeanShell, основанный на Java.

Дэн Леук – директор корпорации Ikayzo с офисами в Токио и Гонолулу, которая занимается интерактивным дизайном и разработкой программного обеспечения. В число клиентов Ikayzo входят Sony, Oracle, Nomura, PIMCO и правительство США. Дэн Леук активно участвует в деятельности Java-сообщества и является председателем нескольких экспертных групп Java Community Process.

В этой книге есть все. Авторы проделали огромную работу.
Я планирую пользоваться этой книгой буквально каждый день.

Кристофер Бейсингер, программист

ISBN 978-5-699-71504-6

9 785699 715046 >

ЭКСМО

O'REILLY®

Материал, защищенный авторским правом

ется. Написать отзыв