

O'REILLY®

5-е издание

Программируем на Java



Марк Лой, Патрик Нимайер, Дэниэл Лук

FIFTH EDITION

Learning Java

*An Introduction to Real-World
Programming with Java*

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javablib>

Marc Loy, Patrick Niemeyer, and Daniel Leuck

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Программируем на Java

5-е международное издание

Марк Лой, Патрик Нимайер, Дэниэл Лук



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1
УДК 004.43
Л68

Лой Марк, Нимайер Патрик, Лук Дэниэл

Л68 Программируем на Java. 5-е межд. изд. — СПб.: Питер, 2023. — 544 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1836-6

Неважно кто вы — разработчик ПО или пользователь — в любом случае слышали о языке Java. В этой книге вы на конкретных примерах изучите основы Java, API, библиотеки классов, приемы и идиомы программирования. Особое внимание авторы уделяют построению реальных приложений.

Вы освоите средства управления ресурсами и исключениями, а также познакомитесь с новыми возможностями языка, появившимися в последних версиях Java.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы являются действующими.

ISBN 978-1492056270 англ.

Authorized Russian translation of the English edition of Learning Java, 5th Edition
ISBN 9781492056270 © 2020 Marc Loy, 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.

ISBN 978-5-4461-1836-6

© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Бестселлеры O'Reilly», 2022

Краткое содержание

Предисловие	16
Глава 1. Современный язык	23
Глава 2. Первое приложение.....	53
Глава 3. Рабочие инструменты.....	90
Глава 4. Язык Java	110
Глава 5. Объекты в Java	154
Глава 6. Обработка ошибок и запись в журнал.....	205
Глава 7. Коллекции и обобщения	236
Глава 8. Текст, числа, дата и время	264
Глава 9. Потоки.....	302
Глава 10. Десктопные приложения	332
Глава 11. Сетевые коммуникации и ввод-вывод.....	390
Глава 12. Веб-программирование	451
Глава 13. Перспективы Java	486
Приложение. Примеры кода и IntelliJ IDEA	497
Глоссарий.....	513
Об авторах	539
Иллюстрация на обложке	540

Оглавление

Предисловие	16
Кому пригодится эта книга.....	16
Последние изменения	17
Что нового в этом издании книги (Java 11, 12, 13, 14)	18
Структура книги	18
Интернет-ресурсы.....	20
Условные обозначения.....	20
Использование исходного кода примеров.....	21
Благодарности	21
От издательства.....	22
 Глава 1. Современный язык.....	23
Появление Java	24
Происхождение Java	24
Развитие.....	26
Виртуальная машина	27
Сравнение Java с другими языками.....	30
Структурная безопасность.....	34
Упрощать, упрощать, упрощать.	34
Безопасность типов и связывание методов.....	35
Инкрементальная разработка	37
Динамическое управление памятью.....	37
Обработка ошибок.....	39
Потоки.....	39
Масштабируемость.....	40
Безопасность на уровне исполнительной системы Java	41
Верификатор.....	42

Загрузчики классов.....	44
Менеджеры безопасности	44
Безопасность на уровнях приложения и пользователя.....	46
История Java	47
Прошлое: Java 1.0 — Java 13.....	47
Настоящее: Java 14	49
Будущее.....	51
Доступные средства.....	52
Глава 2. Первое приложение	53
Инструменты и среда Java.....	53
Установка JDK.....	54
Установка OpenJDK в Linux	55
Установка OpenJDK в macOS	56
Установка OpenJDK в Windows	57
Настройка конфигурации IntelliJ IDEA и создание проекта.....	61
Запуск программы.....	64
Загрузка примеров кода	64
HelloJava	66
Классы.....	68
Метод main().....	69
Классы и объекты	71
Переменные и типы.....	72
HelloComponent	72
Наследование.....	74
Класс JComponent.....	75
Отношения между классами	75
Пакеты и импортирование.....	76
Метод paintComponent()	78
HelloJava2: продолжение	79
Переменные экземпляра	81
Конструкторы.....	82
События.....	84
Метод repaint()	87
Интерфейсы	87
До свидания... и снова здравствуйте!	89

Глава 3. Рабочие инструменты	90
Среда JDK	90
Виртуальная машина Java	91
Запуск приложений Java	91
Системные параметры	93
Classpath	94
javap	95
Модули	96
Компилятор Java	96
Первые эксперименты с Java	98
JAR-файлы	104
Сжатие	104
Утилита jar	105
Утилита pack200	108
Следующий шаг	109
Глава 4. Язык Java	110
Кодирование текста	111
Комментарии	113
Комментарии javadoc	114
Переменные и константы	116
Типы	118
Примитивные типы	119
Ссылочные типы	124
Автоматическое определение типов	126
Передача ссылок	126
Несколько слов о строках	128
Команды и выражения	128
Команды	129
Выражения	138
Массивы	144
Типы массивов	145
Создание и инициализация массива	146
Использование массивов	148
Анонимные массивы	149

Многомерные массивы.....	150
Типы, классы, массивы... и jshell	152
Глава 5. Объекты в Java	154
Классы.....	155
Объявление классов и создание экземпляров	156
Обращение к полям и методам.....	158
Статические поля и методы	163
Методы	165
Локальные переменные.....	166
Замещение.....	167
Статические методы	169
Инициализация локальных переменных	171
Передача аргументов и ссылки	172
Обертки для примитивных типов	174
Перегрузка методов.....	176
Создание объектов	177
Конструкторы.....	178
Работа с перегруженными конструкторами.....	179
Уничтожение объектов.....	181
Уборка мусора.....	181
Пакеты	183
Импортирование классов.....	183
Пользовательские пакеты	185
Видимость полей и методов класса.....	187
Компиляция с пакетами.....	189
Нетривиальное проектирование классов.....	189
Субклассирование и наследование.....	190
Интерфейсы	195
Внутренние классы	198
Анонимные внутренние классы	200
Систематизация кода и планирование на случай ошибок	202
Глава 6. Обработка ошибок и запись в журнал	205
Исключения	206
Исключения и классы ошибок.....	207

Обработка исключений	209
Всплывающие исключения	212
Трассировка стека	213
Проверяемые и непроверяемые исключения	214
Выдача исключений	215
«Расползание» блока try	219
Секция finally	220
try с ресурсами	221
Обработка исключений и быстродействие	223
Проверочные утверждения	223
Включение и отключение проверочных утверждений	225
Использование проверочных утверждений	225
Журнальный API	227
Общие сведения	227
Уровни вывода	230
Простой пример	231
Конфигурирование журнального API	232
Протоколировщик	234
Быстродействие	234
Исключения в реальном мире	235
Глава 7. Коллекции и обобщения	236
Коллекции	236
Интерфейс Collection	237
Разновидности коллекций	238
Интерфейс Map	240
Ограничения типов	242
Контейнеры	243
Можно ли улучшить контейнеры?	244
Знакомство с обобщениями	245
Несколько слов о типах	248
«Ложки не существует»	249
Стирание типов	250
Необработанные типы	252
Отношения между параметризованными типами	253
Почему List<Date> не является List<Object>?	255

Преобразования типов.....	256
Преобразования между коллекциями и массивами.....	258
Итератор	258
Метод sort().....	260
Приложение: деревья на поле	260
Заключение	263
Глава 8. Текст, числа, дата и время.....	264
Строки.....	264
Создание строк.....	265
Создание строк по другим данным.....	266
Сравнение строк.....	267
Поиск.....	268
Список методов String	269
Создание объектов по строкам.....	271
Разбор примитивных чисел.....	271
Разбиение текста на лексемы.....	272
Регулярные выражения.....	274
Общие сведения о регулярных выражениях	274
API регулярных выражений java.util.regex.....	281
Математические средства.....	286
Класс java.lang.Math.....	287
Большие числа.....	291
Дата и время	292
Локальные дата и время.....	293
Создание и обработка значений даты и времени.....	294
Часовые пояса.....	295
Разбор и форматирование даты и времени.....	296
Разбор ошибок.....	298
Временные метки	300
Другие полезные средства	300
Глава 9. Поток.....	302
Знакомство с потоками.....	303
Класс Thread и интерфейс Runnable	304
Управление потоками	307
Смерть потоков	313

Синхронизация	315
Организация последовательного доступа к методам	316
Обращение к переменным классов и экземпляров из нескольких потоков	321
Планирование и приоритеты	323
Состояние потока	324
Квантование	325
Приоритеты	326
Уступка управления	327
Быстродействие потоков	328
Цена синхронизации	328
Потребление ресурсов потоками	329
Вспомогательные средства параллелизма	330
Глава 10. Desktopные приложения	332
Кнопки, ползунки и текстовые поля	333
Иерархии компонентов	333
Архитектура «Модель — представление — контроллер»	334
Надписи и кнопки	335
Текстовые компоненты	341
Другие компоненты	348
Контейнеры и макеты	353
JFrame и JWindow	353
JPanel	355
Менеджеры макетов	356
События	365
События мыши	365
События действий	369
События изменений	371
Другие события	372
Модальные и всплывающие окна	373
Диалоговые окна сообщений	374
Диалоговые окна подтверждения	377
Диалоговые окна ввода	378

Влияние многопоточности.....	379
SwingUtilities и обновление компонентов	380
Таймеры	383
Что дальше?.....	386
Меню.....	386
Хранение конфигураций.....	388
Нестандартные компоненты и Java 2D	388
JavaFX	389
Думайте о пользователях	389
Глава 11. Сетевые коммуникации и ввод-вывод	390
Потоки данных.....	390
Базовый ввод-вывод	393
Символьные потоки данных.....	395
Обертки для потоков данных.....	397
Класс java.io.File	401
Файловые потоки данных.....	407
RandomAccessFile	410
Файловый API пакета NIO	411
FileSystem и Path.....	411
Файловые операции NIO.....	414
Пакет NIO	417
Асинхронный ввод-вывод.....	418
Быстродействие	419
Отображаемые и блокируемые файлы	419
Каналы	419
Буферы.....	420
Кодировщики и декодировщики символов.....	424
FileChannel	426
Сетевое программирование	430
Сокеты	432
Клиенты и серверы	433
Клиент DateAtHost.....	438
Сетевая игра.....	440
Дальнейшие исследования	450

Глава 12. Веб-программирование	451
URL.....	451
Класс URL.....	452
Потоковые данные.....	454
Получение контента в виде объекта	454
Управление соединениями.....	456
Проблема обработчиков.....	457
Полезные фреймворки обработчиков	458
Взаимодействие с веб-приложениями.....	458
Метод GET	459
Метод POST.....	460
URLConnection	464
HTTPS и безопасная передача данных	464
Веб-приложения Java	464
Жизненный цикл сервлета.....	466
Сервлеты	467
Сервлет HelloClient	469
Ответ сервлета.....	471
Параметры сервлетов	472
Сервлет ShowParameters.....	473
Управление сеансами пользователей	475
Сервлет ShowSession	476
Контейнеры сервлетов.....	479
Настройка конфигурации с использованием web.xml и аннотаций	480
URL-шаблоны	483
Развертывание сервлета HelloClient.....	484
Безграничный интернет.....	485
 Глава 13. Перспективы Java	 486
Выпуски Java	486
JCP и JSR.....	487
Лямбда-выражения	488
Переработка существующего кода	489
За пределами базовых возможностей Java.....	495
Заключение и следующие шаги.....	495

Приложение. Примеры кода и IntelliJ IDEA	497
Где загрузить примеры кода	497
Установка IntelliJ IDEA.....	499
Установка в Linux	499
Установка в macOS	499
Установка в Windows.....	500
Импортирование примеров кода.....	502
Запуск примеров кода	505
Загрузка кода веб-приложений.....	509
Работа с сервлетами.....	510
Глоссарий	513
Об авторах	539
Иллюстрация на обложке.....	540

Предисловие

Эта книга научит вас программировать на языке Java и использовать среду разработки приложений. Если вы разработчик или опытный интернет-пользователь, то наверняка слышали об этом языке. Его появление стало одним из ярчайших событий в истории интернета, а бизнес в интернете вырос до сегодняшнего уровня во многом благодаря Java-приложениям. Вероятно, Java является самым популярным в мире языком программирования. Миллионы разработчиков пишут Java-приложения почти для всех видов компьютеров, которые только можно представить. В отношении спроса на программистов Java превосходит такие языки, как C++ и Visual Basic. Он стал фактическим стандартом для разработки некоторых видов программного обеспечения, особенно веб-сервисов. Многие вузы включают Java в начальные курсы программирования наряду с другими актуальными современными языками. Возможно, вы прямо сейчас читаете эти слова на своих учебных занятиях!

Книга даст вам хорошее представление об основах языка Java, в том числе об интерфейсах программирования приложений (API), библиотеках классов, приемах программирования и идиомах. Мы подробно рассмотрим многие интересные области, а некоторые темы затронем лишь в общих чертах. Другие книги издательства O'Reilly продолжают с того уровня знаний, на котором мы остановимся, и предоставляют более полную информацию о конкретных областях и сферах применения Java.

В подходящих случаях мы будем показывать наглядные, реалистичные и интересные примеры кода, избегая монотонного перечисления возможностей. Эти примеры очень просты, но они подскажут вам, что вы сможете делать с помощью Java самостоятельно. Мы не хотим заниматься на этих страницах разработкой очередного «приложения-бестселлера», а вместо этого постараемся дать вам отправную точку для долгих экспериментов и вдохновить на ваш собственный проект такого масштаба.

Кому пригодится эта книга

Эта книга написана для профессионалов в области информационных технологий, для студентов, технических специалистов и «финских хакеров». Она

будет полезна всем, кому нужен практический опыт работы с Java, особенно с целью создания реальных приложений. Кроме того, книгу можно использовать в качестве экспресс-курса по объектно-ориентированному программированию, сетевым приложениям и пользовательским интерфейсам.

В процессе изучения Java вы освоите эффективный и практичный подход к разработке программного обеспечения, началом которого станет глубокое понимание основ языка Java и его API.

На первый взгляд Java имеет много общего с языками C и C++, и если у вас есть опыт программирования на одном из них, то вам будет проще изучить Java. Но если у вас нет такого опыта, не огорчайтесь. Не надо уделять излишнее внимание синтаксическому сходству между Java и C или C++. Во многих отношениях Java ближе к более динамическим языкам, таким как Smalltalk и Lisp. Хорошо, если вы уже знаете другие объектно-ориентированные языки, но в этом случае вам придется, скорее всего, пересмотреть некоторые представления и изменить некоторые привычки. Считается, что язык Java намного проще таких языков, как C++ и Smalltalk. Если вы хорошо учитеcь на коротких примерах и на собственном опыте, то эта книга должна вам понравиться.

В конце книги мы будем рассматривать Java в контексте веб-приложений, веб-служб и обработки запросов, поэтому вы должны хотя бы в общих чертах знать, как устроены браузеры, серверы и документы.

Последние изменения

Это, пятое, издание книги «Программируем на Java» («Learning Java») можно также считать седьмым, обновленным и переименованным изданием нашей предыдущей популярной книги «Exploring Java». В каждом очередном издании мы старались не только добавлять материал о новых возможностях языка, но и тщательно пересматривать и обновлять весь существующий материал, чтобы систематизировать его и отражать на страницах наш многолетний опыт исследований и практического программирования.

Одно из заметных изменений в последних изданиях книги — сокращение (а затем и удаление) материала о работе с апплетами, которые теперь уже практически не используются при создании интерактивных веб-страниц. С другой стороны, значительно расширены темы веб-приложений и веб-служб Java, ставших вполне зрелыми технологиями.

Мы рассматриваем все важные особенности последней (на момент написания книги) из тех версий Java, которые сопровождаются долгосрочной поддержкой от Oracle. Это Java 11, а ее полное название — Java Standard Edition (SE) 11. (Бесплатный аналог — OpenJDK 11.) Кроме того, мы упоминаем некоторые

особенности трех промежуточных версий: Java 12, Java 13 и Java 14. В компании Sun Microsystems, которая была «хранителем» Java до Oracle, за много лет несколько раз меняли схему нумерации версий. Чтобы подчеркнуть множество ценных возможностей, появившихся в версии Java 1.2, ее обозначили термином «Java 2», а также отказались от термина JDK в пользу SDK. Шестая по порядку версия (следующая после Java 1.4) получила название Java 5.0, и тогда же Sun вернула термин JDK. Только после этого продолжилась обычная нумерация: вышли версии Java 6, Java 7 и т. д.

Сейчас перед нами Java 14. Эта версия представляет собой хорошо развитый язык, в котором появился ряд изменений синтаксиса и обновлений API и библиотек. Мы постарались отразить эти новые возможности в примерах кода, чтобы показать современные приемы и стиль программирования на Java.

Что нового в этом издании книги (Java 11, 12, 13, 14)

Это издание мы по традиции переработали таким образом, чтобы сделать его как можно более полным и актуальным. Мы учли изменения, появившиеся в Java 11 (напомним: это версия с долгосрочной поддержкой), а также в промежуточных версиях: Java 12, 13 и 14. (Подробнее о средствах Java, включенных в последние версии и исключенных из них, рассказано в главе 13.) Мы добавили в это издание следующие темы:

- Новые возможности языка, в том числе автоматическое определение (выведение) типов в обобщениях, усовершенствованная обработка исключений и синтаксис автоматического управления ресурсами.
- Интерактивная среда `jshell` для экспериментов с фрагментами кода.
- Выражения `switch`.
- Лямбда-выражения.
- Обновленные примеры и объяснения по всей книге.

Структура книги

Структура книги выглядит примерно так:

- Главы 1 и 2 содержат введение в концепцию языка Java, а также простейшее руководство, которое поможет вам немедленно приступить к программированию.
- В главе 3 рассматриваются важнейшие инструменты для разработки программ на Java (компилятор, интерпретатор, `jshell` и упаковщик JAR).

- В главах 4 и 5 представлены фундаментальные концепции программирования, после чего описывается сам язык Java. Изложение начинается с базового синтаксиса, а затем переходит к классам и объектам, исключениям, массивам, перечислениям, аннотациям и другим темам.
- В главе 6 рассматриваются исключения, способы обработки ошибок и средства журналирования (логирования).
- В главе 7 рассматриваются коллекции, обобщения и отношения между параметризованными типами Java.
- Глава 8 посвящена обработке текста, математическим вычислениям и некоторым другим средствам базового API.
- В главе 9 рассматриваются средства для создания многопоточных приложений.
- В главе 10 представлены основы разработки графических интерфейсов (GUI) с помощью пакета Swing.
- Глава 11 посвящена вводу-выводу Java, потокам данных, файлам, сокетам, сетям и пакету NIO.
- В главе 12 рассматриваются веб-приложения, сервлеты, WAR-файлы и веб-службы.
- Глава 13 рассказывает о процессе развития Java. Она поможет вам отслеживать будущие изменения в языке и модернизировать существующий код, используя новые возможности (например, лямбда-выражения, впервые представленные в Java 8).

Если вы похожи на нас, то вы не читаете книги с начала до конца. А если вы очень похожи на нас, то наверняка не станете читать это предисловие. Но вдруг вы все-таки с него начнете? На этот случай мы дадим несколько рекомендаций:

- Если вы программист и хотите за пять минут понять всю суть Java, то вас, скорее всего, заинтересуют примеры кода. Для начала просмотрите главу 2. Если она не вызовет энтузиазма, перейдите к главе 3 — там рассказано, как использовать компилятор и интерпретатор. Это станет хорошим первым шагом.
- Если вы собираетесь писать приложения для работы в локальной сети или в интернете, обратитесь к главам 11 и 12. Сетевые функции — это одна из самых интересных и важных частей Java.
- В главе 10 рассматриваются графические средства и компоненты Java. Это важно, если вы собираетесь писать обычные десктопные приложения с графическим интерфейсом (то есть приложения для настольных компьютеров).

- Глава 13 рассказывает о том, как всегда быть в курсе происходящих в языке Java изменений, независимо от того, что именно вас интересует.

Интернет-ресурсы

В интернете есть множество источников с информацией о языке Java. Прежде всего, достоверную информацию вы найдете на официальном сайте Oracle (<https://www.oracle.com/java/technologies>). В частности, Oracle публикует документацию с описаниями классов, методов, операторов и других синтаксических конструкций языка, а также дистрибутивы выпусков Java. Именно с сайта Oracle вам лучше всего загрузить эталонную реализацию JDK, которая включает в себя компилятор, интерпретатор и другие инструменты. Oracle также поддерживает официальный сайт проекта OpenJDK (<https://openjdk.java.net>) — так называется основная версия Java с открытым исходным кодом, в состав которой также входят компилятор, интерпретатор и другие инструменты. Мы будем использовать OpenJDK для всех примеров кода в этой книге.

Условные обозначения

В книге используются следующие шрифтовые обозначения:

Исходный код

Этим шрифтом выделены примеры исходного кода, а также приглашения командной строки и результаты текстового вывода на экран.

Комментарии в исходном коде

Этим шрифтом в исходном коде выделены комментарии.

Команды и заменяемые элементы

Этим шрифтом выделены команды, которые вводятся в командной строке, а также те элементы в исходном коде, которые должны быть заменены читателем.

Служебные слова и символы

Этим шрифтом в основном тексте выделены слова и символы, которые обозначают классы, методы, команды, файлы, пути, форматы, параметры, значения, теги и другие компьютерные сущности.

Термины

Этим шрифтом в основном тексте выделены термины, когда они вводятся впервые, а также важные понятия и названия.

Веб-ссылки

Этим шрифтом в основном тексте выделены адреса веб-сайтов с полезной информацией.

Использование исходного кода примеров

Если у вас возникнут вопросы технического характера по использованию примеров кода, направляйте их по электронной почте на адрес bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Многие люди внесли свой вклад в работу над книгой — как в ее первоначальном варианте «Exploring Java», так и в этом издании. Прежде всего, мы благодарим Тима О'Рейли за то, что он предоставил нам возможность написать эту книгу. Спасибо Майку Лукидесу (Mike Loukides), редактору всей серии; его терпение и опыт постоянно направляют нас на истинный путь. Другие сотрудники O'Reilly, в том числе Амелия Блевинс (Amelia Blevins), Зен Маккуэйд (Zan McQuade), Корбин Коллинз (Corbin Collins) и Джессика Хаберман (Jessica Haberman), неперестанно делились с нами своим опытом и вдохновением. Это

предел наших мечтаний — работать со столь квалифицированной и доброжелательной командой.

Исходная версия глоссария позаимствована из книги Дэвида Фленагана (David Flanagan) «Java in a Nutshell», вышедшей в издательстве O'Reilly. Также из книги Дэвида взяты некоторые диаграммы об иерархии классов. Эти диаграммы построены на основе похожих диаграмм Чарльза Л. Перкинса (Charles L. Perkins).

Мы искренне благодарны Рону Бекеру (Ron Becker) за дельные советы и интересные идеи с точки зрения дилетанта, далекого от мира программирования. Благодарим Джеймса Эллиота (James Elliott) и Дэна Лука (Dan Leuck) за их превосходные и актуальные отзывы о техническом содержании этого издания. Как это часто бывает в мире программирования, взгляд со стороны бесценен. Нам очень повезло, что рядом с нами оказались такие внимательные люди.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Современный язык

Сегодня самые сложные проекты и самые грандиозные возможности для разработчиков связаны с освоением потенциала сетей. Приложения (программы), создаваемые в наши дни, независимо от их масштаба и предполагаемой аудитории, почти наверняка будут работать на устройствах, подключенных к интернету. Сети становятся все более необходимыми, и соответственно меняются требования к привычному программному обеспечению и возникает спрос на совершенно новые виды приложений, список которых стремительно разрастается.

Всем нам нужны такие программные продукты, которые одинаково работают на всех платформах и хорошо совместимы с другими приложениями. Нам нужны динамические приложения, использующие всю мощь глобальной сети и взаимодействующие с разнородными распределенными источниками информации. Нам нужно по-настоящему распределенное ПО, которое легко расширяется и обновляется. Нам нужны «умные» приложения, способные самостоятельно анализировать интернет, выискивать нужную информацию и выполнять функции электронных агентов. Потребность в таких приложениях возникла давно, но они начали появляться только в последние годы.

В прошлом главная проблема была связана с недостаточными возможностями инструментария для создания таких приложений. Требования к скорости и портируемости были по большей части взаимоисключающими, а о безопасности часто забывали или понимали ее неправильно. Полностью портируемые языки были громоздкими, интерпретируемыми и медленными. Но они все равно пользовались популярностью: как из-за своей высокоуровневой функциональности, так и из-за портируемости. В быстрых языках скорость достигалась путем привязки к конкретным платформам, поэтому о полноценной портируемости говорить не приходилось. Существовали и безопасные языки, но они в основном были ответвлениями портируемых языков и имели те же недостатки. В отличие от них, Java — современный язык, в котором сочетаются все три качества: портируемость, скорость и безопасность. Именно поэтому Java занимает доминирующее положение в мире программирования более чем через 20 лет после своего появления.

Появление Java

Язык Java, разработанный в Sun Microsystems под руководством асов сетевых технологий Джеймса Гослинга (James Gosling) и Билла Джоя (Bill Joy), проектировался как машинно-независимый язык программирования, достаточно безопасный для работы в сети и при этом достаточно быстрый для замены низкоуровневого машинного кода. Java решил все названные выше проблемы и сыграл важную роль в развитии интернета, что привело нас к нынешнему положению вещей.

На первых порах энтузиазм по поводу Java был основан прежде всего на возможности создания встраиваемых в веб-страницы приложений, называемых *апплетами* (*applets*). Но в те времена возможности апплетов и клиентских приложений с графическим интерфейсом, написанных на Java, были невелики. Зато теперь в Java есть Swing — полноценный инструментарий для создания графических интерфейсов. В результате язык Java стал хорошей платформой для разработки традиционных клиентских приложений, хотя в этой области у него появилось немало конкурентов.

Но что еще важнее, язык Java стал ведущей платформой для приложений на базе веб-технологий и веб-служб. Эти приложения используют разные технологии, включая API сервлетов Java, веб-службы Java и многие популярные фреймворки и серверы приложений Java — как коммерческие, так и с открытым кодом. Благодаря своей портируемости и скорости, язык Java завоевал репутацию оптимальной платформы для разработки современных бизнес-приложений. Серверы Java, работающие на платформах Linux с открытым кодом, заняли центральное место в деловом и финансовом мире.

Эта книга покажет вам, как использовать Java для решения реальных задач программирования. В последующих главах рассматриваются многие возможности: от обработки текста до сетевых коммуникаций, создания десктопных приложений на базе Swing и облегченных приложений на базе веб-технологий и служб.

Происхождение Java

Фундамент Java заложил в 1990-е годы Билл Джой (Bill Joy) — патриарх и ведущий исследователь компании Sun Microsystems. В то время она конкурировала на относительно небольшом рынке рабочих станций, а компания Microsoft начала доминировать на более массовом рынке персональных компьютеров с процессорами Intel. Когда стало ясно, что Sun опоздала на поезд революции персональных компьютеров, Джой переехал в Аспен (штат Колорадо) для проведения перспективных исследований. Ему нравилась идея решения сложных задач простыми программными средствами, и он основал компанию с подходящим названием Sun Aspen Smallworks.

В небольшую команду программистов, собравшихся в Аспене, одним из первых пришел Джеймс Гослинг (James Gosling), которого называют отцом Java. В начале 1980-х он прославился как автор Gosling Emacs — новой версии текстового редактора Emacs, которую он написал на языке C для операционной системы Unix. Gosling Emacs стал популярным, но вскоре его затмила бесплатная версия под названием GNU Emacs, написанная разработчиком исходной версии Emacs. К тому времени Гослинг переключился на разработку оконной системы Sun NeWS, которая в 1987 году конкурировала с X Window System за позицию графической оболочки Unix. И хотя некоторые считают, что система NeWS превосходила X, она все-таки проиграла, потому что компания Sun сделала ее проприетарной и не стала публиковать исходный код, тогда как первые разработчики X сформировали «Консорциум X» и пошли по противоположному пути.

В ходе работы над NeWS Гослинг осознал всю мощь интеграции выразительного языка с оконным графическим интерфейсом, поддерживающим работу в сети. И в компании Sun поняли, что сообщество интернет-программистов категорически не хочет принимать проприетарные стандарты, даже самые замечательные. Неудача NeWS заложила основу схемы лицензирования Java и открытого кода (хотя до настоящего «open source» дело не дошло). Гослинг принес полученные знания в новый проект Билла Джоя. В 1992 году Sun основала для этого проекта дочернюю компанию FirstPerson. Она должна была вывести Sun на рынок бытовой электроники.

Команда FirstPerson создавала программное обеспечение для таких устройств, как сотовые телефоны и карманные компьютеры (PDA).

Работающие в этих устройствах приложения должны были в реальном времени передавать данные через дешевые инфракрасные интерфейсы и традиционные пакетные сети. Из-за недостатка оперативной памяти и малой пропускной способности каналов приходилось писать компактный и эффективный код. И конечно, приложения такого рода надо было делать безопасными и надежными. Вначале Гослинг с коллегами программировали на C++, но вскоре обнаружили, что для их задач этот язык слишком сложен, неповоротлив и уязвим. Тогда они решили начать с нуля, и Гослинг стал работать над тем, что он называл «C++ минус минус».

Провал Apple Newton (первого карманного компьютера Apple) показал, что время PDA еще не настало. Поэтому в Sun переключили усилия компании FirstPerson на интерактивное телевидение (ITV). Для программирования ТВ-приставок разработали язык Oak — один из ближайших предков Java. Но при всей своей элегантности и возможности безопасного обмена данными язык Oak не смог спасти бесперспективную идею интерактивного телевидения. Оно не интересовало покупателей, и вскоре в Sun отказались от этой концепции.

Тогда Джой и Гослинг объединили усилия в поиске актуальной стратегии применения своего новаторского языка. Дело было в 1993 году, и взрывной рост интереса ко Всемирной паутине открывал новые возможности. Язык Oak был компактным, безопасным, архитектурно-независимым и объектно-ориентированным. Так получилось, что все эти свойства входили в набор требований к универсальному языку программирования для интернета. Компания Sun быстро сменила приоритеты, и после небольшой переработки Oak превратился в Java.

Развитие

Не будет преувеличением сказать, что язык Java (и его вариант, ориентированный на разработчиков, — Java Development Kit, или JDK) распространялся стремительно, как степной пожар. Еще до первого официального выпуска, когда Java не был полноценным программным продуктом, за него ухватились лидеры ИТ-бизнеса. Лицензии на Java получили Microsoft, Intel, IBM и почти все остальные крупные производители компьютеров и программного обеспечения. Тем не менее даже при такой поддержке новый язык испытал в первые годы немало трудностей роста.

Начались нарушения контрактов и антимонопольные судебные процессы между Sun и Microsoft, вызванные использованием Java в браузере Internet Explorer. Это затормозило применение нового языка в Windows — самой популярной в мире десктопной операционной системе. Участие Microsoft в работе над Java также стало предметом крупного федерального иска о недобросовестной конкуренции; свидетели дали в суде показания, что гигантская корпорация намеренно стремилась подорвать Java, внедряя несовместимости в свою версию языка. Тем временем Microsoft представила в рамках инициативы .NET свой собственный язык C#, производный от Java, и передумала включать Java в Windows. C# оказался очень хорошим языком, и за последние годы в нем появилось даже больше инноваций, чем в Java.

Но Java и по сей день продолжает распространяться на многих платформах. С первого взгляда на архитектуру Java становится ясно, что ее самые замечательные возможности основаны на изолированной среде виртуальной машины, в которой выполняются Java-приложения. Язык Java предусмотрительно проектировали таким образом, чтобы можно было реализовать архитектуру его поддержки разными способами: как на программном уровне (в операционных системах), так и на аппаратном уровне (в специальном оборудовании). Аппаратные реализации Java используются в некоторых смарт-картах и в других встроенных системах. Вы можете найти интерпретаторы Java даже в таких «носимых устройствах», как электронные кольца и опознавательные жетоны. А программные реализации Java созданы для всех современных компьютерных

платформ, в том числе мобильных. Один из вариантов Java стал компонентом операционной системы Google Android, на которой работают миллиарды смартфонов и других мобильных устройств.

В 2010 году корпорация Oracle купила Sun Microsystems и стала куратором языка Java. Начало оказалось неудачным: Oracle подала судебный иск против Google из-за интеграции Java в Android — и проиграла. В 2011 году Oracle выпустила Java 7 — важную версию, дополненную новым пакетом ввода-вывода. В 2017 году в состав Java 9 вошли модули для решения давно существующих проблем со списком путей `classpath` и с растущим размером JDK. После выпуска Java 9 начался быстрый процесс обновлений, в результате которого появилась актуальная на момент написания книги версия с долгосрочной поддержкой: Java 11. (Об этих и других версиях подробнее рассказано в разделе «История Java», с. 47.) Корпорация Oracle продолжает руководить разработкой языка, но она поделила мир Java надвое: перевела основную линейку версий Java на дорогостоящую коммерческую лицензию, но сохранила и бесплатный вариант под названием OpenJDK. Он по-прежнему общедоступен, его любят и на него ориентируются многие разработчики.

Виртуальная машина

Язык Java одновременно является компилируемым и интерпретируемым. Сначала компилятор преобразует написанный программистом исходный код в простые двоичные команды, очень похожие на машинный код обычных микропроцессоров. Отличие в том, что исходный код C и C++ компилируется в машинный код для конкретной модели процессора, а исходный код Java компилируется в универсальный формат — *байт-код*, представляющий собой команды для *виртуальной машины (VM)*.

Скомпилированный байт-код выполняется интерпретатором исполнительной системы Java, который делает все то же самое, что обычный аппаратный процессор, но в безопасной виртуальной среде. Он выполняет набор команд на базе стека и управляет памятью подобно операционной системе. Он создает примитивные типы данных, управляет ими, загружает и активирует новые блоки кода по ссылкам. Важнее всего, что все это делается в соответствии со строго определенной общедоступной спецификацией, по которой любой разработчик может создать Java-совместимую виртуальную машину. Сочетание виртуальной машины и определения языка образует полную спецификацию. В базовом языке Java нет никаких деталей, которые остались бы неопределенными или зависели бы от конкретной реализации. Например, Java строго регламентирует размеры и математические свойства всех своих примитивных типов данных, не оставляя их на усмотрение разработчиков версий для разных платформ.

Интерпретатор Java относительно легок и компактен; его можно реализовать в любой форме, подходящей для конкретной платформы. Интерпретатор может быть отдельным приложением или встроенным компонентом другого приложения (например, веб-браузера). Таким образом, код Java является портируемым по своей природе. Один и тот же байт-код приложения будет правильно работать на любой платформе, в которой есть исполнительная среда Java (рис. 1.1). Вам не придется писать альтернативные версии своего приложения для разных платформ или распространять исходный код среди конечных пользователей.

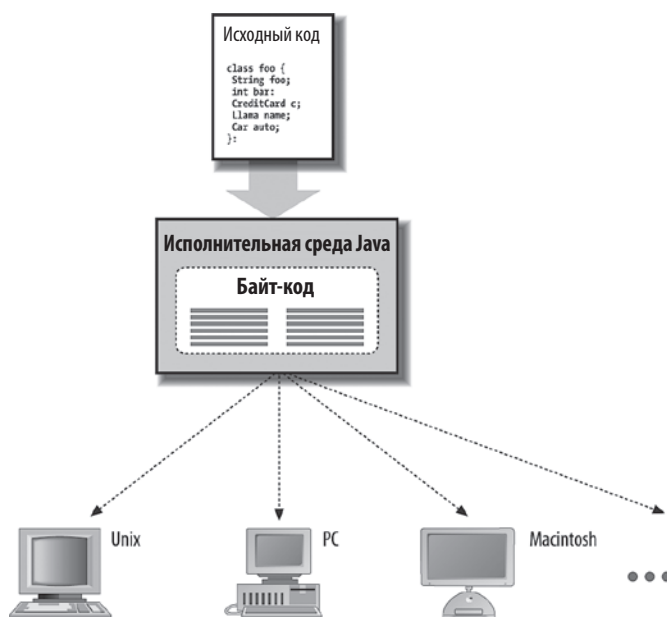


Рис. 1.1. Исполнительная среда Java

Фундаментальной единицей кода Java является *класс*. Как и в других объектно-ориентированных языках, классы представляют собой компоненты приложений, содержащие исполняемый код и данные. Скомпилированные классы Java распространяются (публикуются) в универсальном двоичном формате, который содержит байт-код Java и другую информацию классов. Разработчик может заниматься созданием и сопровождением отдельных классов, сохраняя их в файлах и архивах — локально или на сетевом сервере. Java умеет находить и загружать нужные классы динамически, когда они требуются работающему приложению.

Кроме платформенно-зависимой исполнительной системы, в Java входит ряд фундаментальных классов, содержащих архитектурно-зависимые методы. Эти *платформенные методы* (*native methods*) образуют своего рода шлюз между

виртуальной машиной Java и реальным миром. Они реализуются на языке, компилируемом в платформенный код на платформе размещения, и предоставляют низкоуровневый доступ к ресурсам компьютера: к сети, к оконной системе, к файловой системе и т. д. Однако основная часть кода Java написана на самом языке Java (инициализируемом этими базовыми примитивами), поэтому легко портируется. Сюда входят такие фундаментальные средства Java, как компилятор Java, поддержка сети и библиотек GUI, которые также написаны на Java, поэтому доступны в абсолютно одинаковом виде на всех платформах Java без портирования.

Исторически считалось, что интерпретаторы работают медленно, но Java не является традиционным интерпретируемым языком. Кроме компиляции исходного кода в портируемый байт-код, при разработке языка Java специально предусматривалось, чтобы программные реализации исполнительной системы могли дополнительно оптимизировать свое быстродействие методом компиляции байт-кода в платформенный машинный код «на ходу». Этот механизм называется *динамической компиляцией*, или *JIT-компиляцией* (Just In Time). Благодаря JIT-компиляции, код Java может выполняться с такой же скоростью, как платформенный код, без ущерба для мобильности и безопасности.

Этот момент часто создает недоразумения при сравнении быстродействия языков. Существует только одна причина внутреннего снижения производительности, присущая скомпилированному коду Java на стадии выполнения и необходимая для безопасности и эффективности архитектуры виртуальной машины: проверка границ массивов. Все остальное может оптимизироваться в платформенный код точно так же, как в языках со статической компиляцией. Кроме того, язык Java содержит больше структурной информации, чем многие другие языки; эта информация расширяет возможности оптимизации. Также надо помнить, что оптимизация может применяться на стадии выполнения с учетом фактической логики работы и характеристик приложения. Что можно хорошо сделать на стадии компиляции, но нельзя сделать еще лучше на стадии выполнения? Тут все определяется временем.

У традиционной JIT-компиляции есть проблема: оптимизация кода требует времени. Таким образом, JIT-компилятор может выдавать отличные результаты, но за это приходится расплачиваться заметной задержкой при запуске приложения. Обычно это не создает проблем для приложений, долго работающих на стороне сервера, но может стать серьезной проблемой для клиентских программ и приложений, работающих на компактных устройствах со скромными возможностями. Для решения этой проблемы технология компилятора Java, называемая HotSpot, использует прием *адаптивной компиляции*. Если проанализировать фактическое распределение времени выполнения типичной программы, то станет ясно: почти все время тратится на то, чтобы относительно малые части кода выполнялись снова и снова. Многократно выполняемые блоки кода могут составлять малую часть от размера программы, но такое поведение определяет

ее общее быстродействие. Адаптивная компиляция также позволяет исполнительной среде Java применять новые виды оптимизации, попросту невозможные в языках со статической компиляцией; отсюда и утверждение, что код Java в некоторых случаях может выполняться быстрее кода C или C++.

Чтобы воспользоваться этим фактом, HotSpot вначале работает как обычный интерпретатор байт-кода Java, но с небольшим отличием: он измеряет время выполнения кода («профилирует» код), чтобы определить, какие его части выполняются многократно. Определив, какие части кода критичны для быстродействия, HotSpot компилирует эти секции в оптимальный платформенный машинный код. Поскольку в машинный код компилируется лишь небольшая часть программы, затраты времени на оптимизацию этой части оказываются приемлемыми. Оставшаяся часть программы может вообще не компилироваться (а только интерпретироваться) для экономии памяти и времени. В сущности, виртуальная машина Java может работать в любом из двух режимов: клиентском или серверном. Режим определяет, чему будет отдано предпочтение — скорости запуска и экономии памяти или общему быстродействию. В Java 9 также можно воспользоваться опережающей (Ahead-of-Time, AOT) компиляцией, если минимизация времени запуска вашего приложения действительно важна.

Возникает естественный вопрос: зачем удалять всю полезную информацию профилирования при каждом закрытии приложения? Вообще говоря, компания Sun частично уделила внимание этой теме в Java 5.0 — там появилась возможность использования общих классов, доступных только для чтения, которые хранятся в оптимизированной форме. Тем самым значительно сокращается как время запуска, так и непроизводительные затраты ресурсов на выполнение многих приложений Java на обычной машине. Технология, применяемая для достижения этой цели, сложна, но ее идея проста: оптимизировать только те части программы, которые должны выполняться быстро, и не беспокоиться обо всем остальном.

Сравнение Java с другими языками

Проектируя Java, его создатели руководствовались многолетним опытом программирования на других языках. Уделим немного времени сравнению Java с другими языками на концептуальном уровне — это будет полезно и читателям, уже умеющим программировать, и новичкам, которые хотят увидеть общую картину. Обратите внимание, что эта книга не требует от вас знания какого-либо конкретного языка. Мы надеемся, что все упоминания других языков будут достаточно понятными.

В основании любого универсального языка программирования стоят три столпа: портируемость, скорость и безопасность. На рис. 1.2 показаны результаты сравнения Java с некоторыми языками, популярными на момент его создания.

Вероятно, вы слышали, что у Java много общего с языками C и C++. На самом деле это не так, разве что на самом поверхностном уровне. При первом взгляде на код Java вы заметите, что базовый синтаксис напоминает C или C++... но на этом сходство заканчивается. Java ни в коем случае не является прямым потомком C или относящегося к следующему поколению C++. Сравнивая языки по их свойствам, вы увидите, что у Java больше общего с динамическими языками, такими как Smalltalk и Lisp. Практическая реализация Java в компьютере настолько далека от классического C, что трудно даже представить.

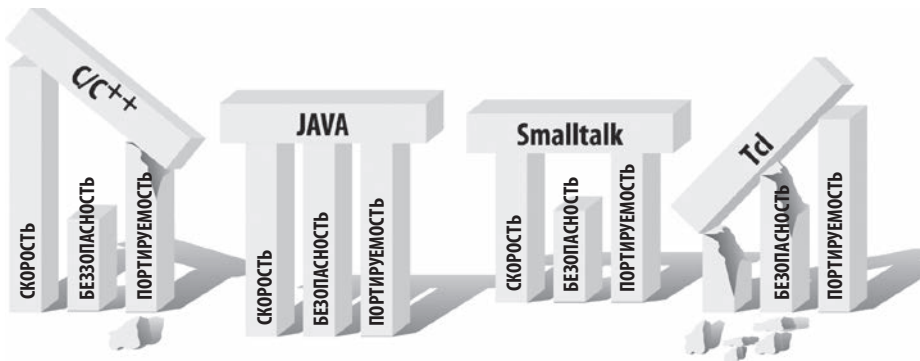


Рис. 1.2. Сравнение языков программирования

Читатели, знакомые с современной ситуацией в программировании, заметят, что в этом сравнении отсутствует такой популярный язык, как C#. Он в значительной мере стал ответом Microsoft на появление Java, и надо признать, что в C# добавлен ряд полезных возможностей. Вследствие общих целей и подходов к проектированию приложений (виртуальная машина, байт-код, изолированная среда и т. д.) эти две платформы не очень существенно различаются по скорости и по характеристикам безопасности. C# обладает примерно такой же портируемостью, как и Java. C# многое позаимствовал из синтаксиса C, но на самом деле является более близким родственником динамических языков. Многие Java-разработчики относительно легко осваивают C#, и наоборот. Большая часть времени, затраченного на переход с одного языка на другой, уходит на изучение стандартной библиотеки.

Тем не менее даже поверхностное сходство между этими языками заслуживает внимания. Java заимствует многое из синтаксиса C и C++, так что вы увидите многие лаконичные языковые конструкции с обилием фигурных скобок и точек с запятой. Java следует философии языка C в том, что хороший язык должен быть компактным; иначе говоря, достаточно кратким и систематичным, чтобы программист мог одновременно удерживать в голове все его возможности. Подобно тому как C можно расширять библиотеками, к основным компонентам Java можно добавлять пакеты классов, расширяющие синтаксис.

Язык С стал таким успешным, потому что он предоставляет достаточно функциональную среду программирования с высокой производительностью и приемлемым уровнем портируемости. Java также старается поддерживать баланс между функциональностью, скоростью и портируемостью, но делает это совершенно иным способом. Язык С жертвует функциональностью ради портируемости; Java на первых порах жертвовал скоростью ради портируемости. Кроме того, в Java решены проблемы безопасности, которые остались в С (хотя в современных компьютерах многие из них решаются в операционной системе и на аппаратном уровне).

Много лет назад, до появления JIT и адаптивной компиляции, Java был медленнее, чем языки со статической компиляцией. Критики постоянно говорили, что он никогда их не догонит. Но как мы рассказали в предыдущем разделе, теперь Java уже сравним по быстродействию с языками С и С++ в эквивалентных задачах, поэтому критики в основном притихли. Движок видеоигры Quake 2 от ID Software, распространяемый с открытым кодом, был портирован на Java. Если Java достаточно быстр даже для шутеров с видом от первого лица, то он наверняка будет достаточно быстрым и для большинства бизнес-приложений.

Языки сценариев, такие как Perl, Python и Ruby, не утратили популярности. Нет причин, по которым язык сценариев не подходил бы для безопасных сетевых приложений. Но как правило, языки сценариев не подходят для серьезного крупномасштабного программирования. Главное преимущество языков сценариев заключается в их динамичности; это мощные инструменты для быстрой разработки. Некоторые языки сценариев (такие, как Perl) также предоставляют мощные средства обработки текста, которые в языках более общего назначения становятся неудобными. Языки сценариев также отличаются высоким уровнем портируемости, хотя бы на уровне исходного кода.

JavaScript (не путайте с Java!) — это объектно-базированный язык сценариев, который компания Netscape много лет назад разработала для своего браузера. Сегодня он является встроенным компонентом большинства браузеров, его используют для динамичных и интерактивных веб-приложений. Название JavaScript происходит от интеграции и некоторого сходства с Java, но по своей сути эти языки совершенно разные. Впрочем, у JavaScript есть важные применения и за пределами браузеров (например, платформа Node.js¹), а его популярность среди разработчиков в разных областях продолжает расти. За дополнительной информацией о JavaScript обращайтесь к книге Дэвида Фленагана (David Flanagan) «JavaScript: The Definitive Guide» (издательство O'Reilly).

Основная проблема языков сценариев — их неформальное отношение к структуре программы и к типам данных. Большинство языков сценариев не явля-

¹ Если вас интересует Node.js, рекомендуем книгу: *Пауэрс III. Изучаем Node.js*. — СПб.: Питер.

ются объектно-ориентированными. В них упрощены системы типов данных, и обычно в них нет проработанной системы видимости переменных и функций. Из-за этих особенностей они хуже подходят для создания больших модульных приложений. Другая проблема языков сценариев — это скорость; из-за своей высокоуровневой природы такие языки, обычно интерпретируемые на уровне исходного кода, часто отличаются медлительностью.

Сторонники конкретных языков сценариев не согласятся с некоторыми из этих обобщений — и в каких-то случаях будут правы. Языки сценариев в последние годы совершенствовались, особенно JavaScript, который стал быстрее в результате колоссальных исследований. Но нельзя отрицать фундаментальный факт: языки сценариев рождались как неформальные, менее структурированные альтернативы языкам системного программирования, и по многим причинам языки сценариев, как правило, плохо подходят для больших или сложных проектов (по крайней мере на данный момент).

Java обладает некоторыми важнейшими преимуществами языков сценариев, прежде всего высокой динамичностью, а также дополнительными преимуществами низкоуровневого языка. В Java есть мощный API регулярных выражений, способный конкурировать с Perl в области работы с текстом, а также есть языковые средства, упрощающие работу с коллекциями, переменными списками аргументов, статическим импортированием методов и другими удобными синтаксическими конструкциями, которые делают код более лаконичным.

Инкрементальная разработка на Java с объектно-ориентированными компонентами позволяет быстро создавать приложения и легко изменять их, тем более что этот язык отличается простотой. Исследования показали, что разработка на Java выполняется быстрее, чем на C или C++, в основном благодаря богатым синтаксическим возможностям¹. Java также включает большую базу стандартных фундаментальных классов для решения таких типичных задач, как создание графических интерфейсов и сетевых приложений. Вы можете использовать Maven Central — внешний ресурс с огромным набором библиотек и пакетов, быстро подключая их к вашей среде программирования для решения всевозможных задач. В дополнение ко всему перечисленному Java обладает масштабируемостью и технологическими преимуществами более статических языков. Java предоставляет безопасную структуру, на базе которой создаются фреймворки более высокого уровня (и даже другие языки).

Мы уже отмечали, что Java по своей архитектуре близок к таким языкам, как Smalltalk и Lisp. Только они используются в основном как исследовательские инструменты, а не средства для разработки крупномасштабных систем. Одна из причин заключается в том, что для этих языков так и не появились стан-

¹ Например, см.: *Phipps G. Comparing Observed Bug and Productivity Rates for Java and C++. Software — Practice & Experience*, том 29, 1999 г.

дартные портируемые интерфейсы с сервисами операционной системы, такие как стандартная библиотека C или фундаментальные классы Java. Исходный код Smalltalk компилируется в формат интерпретируемого байт-кода и может динамически компилироваться в платформенный код «на ходу» — примерно так же, как в Java. Но в Java эта архитектура улучшена: правильность скомпилированного байт-кода проверяется верификатором. Верификатор создает для Java преимущество перед Smalltalk, потому что код Java требует меньшего количества проверок на стадии выполнения. Кроме того, верификатор байт-кода Java помогает решать те проблемы безопасности, которые не решаются в Smalltalk.

Далее в этой главе приводится общий обзор языка Java. Мы расскажем, что нового появилось в Java, что осталось неизменным и почему.

Структурная безопасность

Вы уже знаете, что язык Java проектировался как безопасный. Но что имеется в виду под безопасностью? Безопасность — от чего или от кого? Наибольшее внимание к Java привлекают те средства безопасности, которые дают возможность создания новых типов программ с динамической портируемостью. Java предоставляет несколько уровней защиты от опасных ошибок кода, а также от таких вредоносных объектов, как вирусы и трояны. В следующем разделе будет показано, как архитектура виртуальной машины Java оценивает безопасность кода перед его выполнением и как *загрузчик классов* Java (механизм загрузки байт-кода в интерпретаторе Java) строит «стены» вокруг ненадежных классов. Эти средства закладывают основу для высокоуровневых политик безопасности, которые могут разрешать или запрещать определенные действия на уровне отдельных приложений.

В этом разделе рассматриваются некоторые общие свойства языка Java. По сравнению со специальными средствами безопасности (кстати, часто упускаемыми из виду) еще важнее та безопасность, которую Java обеспечивает благодаря отсутствию многих типичных проблем архитектуры и программирования. В Java есть максимально возможная защита от элементарных ошибок, которые допускают программисты, в том числе при наследовании от старого кода. Создатели Java всегда старались сделать язык простым, включить в него средства, доказавшие свою полезность, и дать программистам возможность создавать на базе языка сложные конструкции, когда возникает такая необходимость.

Упрощать, упрощать, упрощать...

В Java правит простота. Поскольку работа над Java начиналась с нуля, разработчикам удалось избежать тех особенностей, которые оказались проблемными

или неоднозначными в других языках. Например, Java не допускает перегрузки операторов (которая в других языках позволяет программисту переопределять смысл таких знаков, как + и -). В Java нет препроцессора исходного кода, поэтому нет и таких средств, как макросы, команды `#define` или условная компиляция. Эти конструкции существуют в других языках прежде всего для поддержки платформенных зависимостей, и в этом смысле они не нужны в Java. Условная компиляция также часто применяется при отладке, но хорошо проработанные технологии оптимизации исполнительной системы Java и такие средства, как *проверочные утверждения* (*assertions*), решают проблему более элегантно. (Вам определенно стоит заняться их изучением после того, как вы освоите основы программирования на Java.)

Java предоставляет четко определенную структуру *пакетов*, в которых упорядочены файлы классов. Система пакетов позволяет компилятору реализовать часть функциональности традиционной утилиты `make` (программа для получения исполняемых файлов из исходного кода). Компилятор также может работать непосредственно со скомпилированными классами Java, потому что в них сохраняется вся информация о типах; лишние «заголовочные» файлы как в C/C++ не нужны. Все это означает, что для понимания кода Java требуется меньше контекстной информации. Иногда вам будет проще прочитать исходный код Java, чем документацию класса.

Java иначе подходит к некоторым принципам структурирования кода, которые вызывают проблемы в других языках. Например, иерархия классов в Java строится только путем одиночного наследования (у каждого класса может быть только один «родительский» класс), зато разрешено множественное наследование интерфейсов. *Интерфейс* (аналог абстрактного класса в C++) задает логику работы объекта, но не определяет его реализацию. Это чрезвычайно мощный механизм, позволяющий разработчику описать нужную логику работы объекта в виде «контракта», который затем будет действовать и использоваться независимо от любых конкретных реализаций объекта. Интерфейсы в Java устраняют необходимость во множественном наследовании классов и все связанные с ним проблемы.

Как будет показано в главе 4, Java — удивительно простой и элегантный язык программирования, и это одна из важных причин его популярности.

Безопасность типов и связывание методов

Одна из главных характеристик любого языка программирования — тот способ, которым в нем осуществляется *проверка типов*. В общем случае языки делятся на *статические* и *динамические*. Разница между ними определяется объемом информации о переменных, известной на стадии компиляции, относительно информации, известной на стадии выполнения.

В языках со строгой статической типизацией (таких, как C и C++) все типы данных жестко фиксируются на момент компиляции исходного кода. Компилятор извлекает пользу из этого факта, так как у него имеется достаточно информации для выявления многих видов ошибок перед выполнением кода. Например, компилятор не позволит сохранить вещественное значение в целочисленной переменной. Код не требует проверки типов во время выполнения, поэтому он компилируется в компактную и быструю форму. Однако языкам со статической типизацией не хватает гибкости. Они не поддерживают коллекции таким же естественным образом, как языки с динамической типизацией, и в них приложение не может безопасно импортировать новые типы данных во время выполнения.

С другой стороны, в динамических языках (таких, как Smalltalk или Lisp) есть исполнительная система, которая управляет типами объектов и выполняет необходимую проверку типов во время выполнения приложения. Такие языки позволяют реализовать более сложную логику работы и во многих отношениях оказываются более мощными. Но они обычно работают медленнее, менее безопасны и создают больше сложностей при отладке.

Различия между языками сравнивались с различиями между моделями автомобилей¹. Языки со статической типизацией (такие, как C++) напоминают спортивный автомобиль: достаточно быстрый и безопасный, но практичный только в том случае, если вы едете по хорошей асфальтированной дороге. Языки с ярко выраженной динамической типизацией (такие, как Smalltalk) больше напоминают внедорожники: они предоставляют больше свободы, но иногда ими трудно управлять. На них веселее (а иногда и быстрее) проехать напрямую по перелеску, но там можно застрять в канаве или попасть в лапы медведям.

Еще одна характеристика языка — способ связывания вызовов методов с их определениями. В статических языках (таких, как C и C++) связывание методов обычно происходит во время компиляции, если только программист не потребует обратного. С другой стороны, такие языки, как Smalltalk, называются языками с *поздним связыванием*, потому что поиск определений методов осуществляется динамически во время выполнения. Раннее связывание важно по соображениям быстродействия; приложение может выполняться без лишних затрат, вызванных поиском методов во время выполнения. А позднее связывание обеспечивает большую гибкость. Оно также необходимо для таких объектно-ориентированных языков, в которых новые типы могут загружаться динамически и только исполнительная система может определить, какой метод надо вызвать в каждом конкретном случае.

Java обладает некоторыми сильными сторонами как C++, так и Smalltalk; это язык со статической типизацией и поздним связыванием. Каждый объект в Java

¹ Эту аналогию предложил Маршалл П. Клайн (Marshall P. Cline), автор C++ FAQ.

имеет четко определенный тип, известный во время компиляции. Это означает, что компилятор Java может проводить такие же статические проверки типов и анализ использования, как в C++. В результате вы не сможете присвоить объект переменной неправильного типа или вызвать для объекта несуществующие методы. Более того, компилятор Java не позволяет использовать неинициализированные переменные и создавать недоступные команды (см. главу 4).

Однако Java также обладает полноценной динамической типизацией. Исполнительная система Java отслеживает все объекты и позволяет определять их типы и отношения между ними во время выполнения. Таким образом, вы можете проанализировать объект на стадии выполнения, чтобы определить, что он собой представляет. В отличие от C и C++, преобразования объекта от одного типа к другому проверяются исполнительной системой, а новые виды динамически загружаемых объектов могут использоваться с некоторой степенью безопасности типов. Поскольку Java является языком с поздним связыванием, субкласс может переопределять методы своего суперкласса, даже если субкласс загружается во время выполнения.

Инкрементальная разработка

Java переносит всю информацию о типах данных и сигнатурах методов из исходного кода в форму скомпилированного байт-кода. Благодаря этому становится возможной инкрементальная разработка классов Java. Ваш исходный код Java может безопасно компилироваться вместе с классами из других источников, совершенно неизвестных вашему компилятору. Иначе говоря, вы можете писать новый код, который работает с двоичными файлами классов без потери безопасности типов, даже если у вас нет исходного кода этих классов.

Java не страдает от проблемы «хрупкости базовых классов». В таких языках, как C++, разработка базового класса иногда останавливается из-за того, что он имеет множество производных классов; для внесения изменений в базовый класс пришлось бы перекомпилировать все производные классы. Эта проблема особенно сложна для разработчиков библиотек классов. Java обходит эту проблему динамическим поиском полей в классах. Пока класс сохраняет допустимую форму своей исходной структуры, он может развиваться без нарушения работоспособности других классов, производных от него или использующих его.

Динамическое управление памятью

Некоторые важные различия между Java и низкоуровневыми языками (такими, как C и C++) связаны с тем, как реализовано управление памятью. В Java запрещены так называемые указатели, способные ссылаться на произвольные области памяти; зато добавлен механизм уборки мусора и высокоуровневые массивы.

Эти особенности устранили многие проблемы с безопасностью, портируемостью и оптимизацией, не решаемые другими способами.

Уборка мусора спасла бесчисленных разработчиков от самого распространенного источника ошибок программирования на С и С++: явного выделения и освобождения памяти. Исполнительная система Java не только размещает объекты в памяти, но и отслеживает все ссылки на них. Когда объект перестает использоваться, Java автоматически удаляет его из памяти. Как правило, на объекты, ставшие ненужными, можно не обращать внимания: интерпретатор уберет их в подходящий момент.

В Java есть сложный уборщик мусора, который работает в фоновом режиме; это означает, что большинство операций уборки мусора происходит в моменты бездействия, в паузах ввода-вывода, между щелчками мышью или нажатиями клавиш. Эффективные технологии исполнительной системы, такие как HotSpot, обеспечивают очень рациональную уборку мусора, в которой учитываются закономерности использования объектов (например, объектов с малым и большим сроком жизни) и оптимизируется их уничтожение. Исполнительная система Java умеет автоматически настраиваться для оптимального распределения памяти в разных видах приложений, в зависимости от их поведения. Благодаря профилированию на стадии выполнения, автоматическое управление памятью может работать намного быстрее самого аккуратного «ручного» управления — это факт, в который программисты старой школы все еще не могут поверить.

Мы уже отметили, что в Java нет указателей. Строго говоря, это утверждение истинно, но оно может вызвать недоразумения. В Java поддерживаются *ссылки* (*references*), которые представляют собой безопасную разновидность указателей. Ссылка — это строго типизованный идентификатор объекта. В Java обращение ко всем объектам, кроме примитивных числовых типов, осуществляется по ссылкам. Вы можете использовать ссылки для создания всех обычных структур, которые программисты на языке С привыкли создавать с помощью указателей: связанных списков, деревьев и т. д. Но есть одно отличие: со ссылками вы можете это делать только при соблюдении безопасности типов.

Другое важное отличие указателей от ссылок заключается в том, что программист не может выполнять арифметические операции со ссылками для изменения их значений; ссылки указывают только на конкретные методы, объекты или элементы массива. Ссылки атомарны; со значением ссылки нельзя выполнять другие операции, кроме присваивания объекту. Ссылки передаются по значению, и к объекту нельзя обращаться более чем через один уровень косвенности. Защита ссылок — один из самых фундаментальных аспектов безопасности Java. Отсюда следует, что код Java должен «играть по правилам»; он не может залезть туда, куда ему залезать не положено, и он не может обходить правила.

Наконец, надо упомянуть, что массивы в Java являются настоящими, полноценными объектами. Их можно динамически создавать и присваивать, как и любые другие объекты. Массивы знают свои размеры и типы. Хотя вы не можете напрямую определять классы массивов и создавать субклассы, у массивов есть четко определенные отношения наследования, основанные на отношениях их базовых типов. Когда в языке имеются настолько полноценные массивы, у вас практически исчезает необходимость в арифметических операциях с указателями как в C и C++.

Обработка ошибок

Язык Java уходит корнями к сетевым устройствам и встроенным системам. В этих областях важно иметь надежный и разумный способ контроля возникающих ошибок. Поэтому в Java есть мощный механизм исключений, напоминающий аналогичные механизмы в новых реализациях C++. Исключения — это очень естественный и элегантный способ обработки ошибок. Исключения позволяют отделить код обработки ошибок от основного кода, что делает приложения более стабильными, а их логику — более простой для понимания.

Как только в выполняемом приложении возникает исключение (вследствие какой-то ошибки), управление передается в заранее подготовленный блок кода `catch`. Этот блок получает исключение в виде объекта, содержащего информацию о ситуации, которая стала причиной исключения. Компилятор Java требует, чтобы каждый метод либо объявлял исключения, которые он может генерировать, либо перехватывал и обрабатывал их самостоятельно. Таким образом, информация об ошибках является столь же важной, как аргументы и возвращаемые типы методов. Программируя на Java, вы точно знаете, с какими исключительными ситуациями вам придется иметь дело, а компилятор помогает вам писать надежные приложения, не оставляющие эти ситуации необработанными.

Потоки

Современным приложениям необходима высокая степень параллелизма. Даже самое линейное приложение может иметь сложный пользовательский интерфейс, а это означает необходимость в нескольких параллельных процессах. В наши дни компьютеры должны работать быстро, и никому не нравится, когда они начинают «тормозить» на второстепенных задачах, отнимая время. *Потоки (threads)*, также называемые потоками выполнения (в отличие от потоков данных), позволяют эффективно распределять и параллельно выполнять несколько задач, ускоряя работу как клиентских, так и серверных

приложений. В Java работать с потоками несложно, потому что их поддержка встроена в язык.

Параллелизм очень полезен, но для программирования потоков недостаточно выполнять в них разные задачи в одно и то же время. В большинстве случаев потоки должны быть *синхронизированы* (скоординированы) между собой, что было бы трудно без явной поддержки со стороны языка. Java поддерживает синхронизацию на базе модели *мониторов* и *условий* — своего рода системы «ключей и замков» для обращения к ресурсам. Ключевое слово `synchronized` помечает методы и блоки кода для безопасного последовательного доступа к ним внутри объекта. Также предусмотрены простые, примитивные методы для явного ожидания своей очереди и для передачи сигналов между потоками, заинтересованными в одном объекте.

Кроме того, в Java есть высокоуровневый пакет параллелизма, предоставляющий мощные средства для реализации типовых паттернов многопоточного программирования, таких как пулы потоков, координация задач и сложная блокировка. Благодаря этому пакету и сопутствующим инструментам, Java считается одним из лучших языков для работы с потоками.

Некоторым разработчикам никогда не приходится писать многопоточный код, но все-таки работа с потоками считается важным навыком программирования на Java, которым должен владеть каждый разработчик. Эта тема рассматривается в главе 9.

Масштабируемость

На самом низком уровне программы Java состоят из *классов*. Обычно классы представляют собой небольшие модульные компоненты. Кроме классов, Java поддерживает *пакеты* — это тот уровень структуры, на котором классы группируются в функциональные единицы. Пакеты предоставляют схему формирования имен для упорядочения классов, а также второй уровень организационного управления видимостью переменных и методов в приложениях Java.

Внутри пакета класс либо обладает общедоступной видимостью, либо защищен от внешнего доступа. Пакеты формируют другой тип видимости, более близкий к уровню приложения. Этот уровень хорошо подходит для создания пригодных для повторного использования компонентов, совместно работающих в системе. Пакеты также упрощают создание масштабируемых приложений, которые можно развивать, не превращая их в дебри запутанного кода. Для повторного использования кода и масштабирования приложений наиболее эффективна модульная система (появившаяся в Java 9), но эта тема выходит за рамки книги. Модулям полностью посвящена книга Пола Беккера (Paul Bakker) и Сандера Мака (Sander Mak) «Java 9 Modularity» (издательство O'Reilly).

Безопасность на уровне исполнительный системы Java

Одно дело — создать язык, который мешает вам «выстрелить себе в ногу», и совсем другое — создать язык, который помешает кому-то другому «выстрелить вам в ногу».

Инкапсуляция — это концепция сокрытия внутри класса его данных и логики работы. Инкапсуляция является важной частью объектно-ориентированных архитектур, так как помогает писать чистый, состоящий из модулей код. Впрочем, в большинстве языков видимость элементов данных является всего лишь частью отношений между программистом и компилятором. Это вопрос семантики, а не утверждение о фактической безопасности данных в той среде, где выполняется программа.

Когда Бьёрн Страуструп выбрал ключевое слово `private` для обозначения скрытых компонентов классов в C++, он думал, скорее всего, о том, чтобы разработчика не беспокоили запутанные подробности кода других разработчиков, а не о том, как защищать классы и объекты от атак вирусов и троянов. Произвольные преобразования типов и арифметические операции с указателями в C и C++ позволяют легко нарушать разрешения доступа к классам, не нарушая при этом правила языка. Возьмем следующий код:

```
// Код C++
class Finances {
    private:
        char creditCardNumber[16];
        ...
};

main() {
    Finances finances;
    // Формирование указателя для получения доступа
    // к конфиденциальным данным внутри класса
    char *cardno = (char *)&finances;
    printf("Card Number = %.16s\n", cardno);
}
```

В этой маленькой драме на C++ мы написали код, который нарушает инкапсуляцию класса `Finances` и извлекает секретную информацию. Подобные фокусы — злоупотребления с нетипизованными указателями — в Java невозможны. Если этот пример кажется вам нереалистичным, подумайте, как важно защищать фундаментальные (системные) классы исполнительный системы от подобных атак. Если ненадежный код сможет повредить компоненты, предоставляющие доступ к реальным ресурсам (к файловой системе, к сети, к оконной системе), то у него появится возможность перехватить номера ваших кредиток.

В тех случаях, когда приложение Java динамически загружает код из ненадежных источников в интернете и выполняет его одновременно с приложениями, содержащими конфиденциальную информацию, защита должна быть очень глубокой. Модель безопасности Java создает вокруг импортированных классов три уровня защиты (рис. 1.3).



Рис. 1.3. Модель безопасности Java

На внешнем контуре этой системы все решения безопасности на уровне приложения принимаются менеджером безопасности, в сочетании с гибкой политикой безопасности. Менеджер безопасности управляет доступом к системным ресурсам: к файловой системе, к сетевым портам, к оконной среде и т. д. Работа менеджера безопасности зависит от способности загрузчика классов защищать базовые системные классы. Загрузчик классов обеспечивает загрузку классов из локального хранилища или из сети. А на самом внутреннем уровне вся безопасность системы в конечном счете зависит от верификатора, гарантирующего целостность получаемых классов.

Этот верификатор байт-кода Java является специальным модулем и неотъемлемой частью исполнительной системы Java. Однако загрузчики классов и менеджеры безопасности (а точнее, *политики безопасности*) представляют собой компоненты, которые могут быть по-разному реализованы в разных приложениях (например, в серверах и браузерах). Для безопасности в среде Java необходимо правильное функционирование всех этих компонентов.

Верификатор

Первая линия защиты Java — *верификатор байт-кода*. Верификатор читает байт-код перед его выполнением и убеждается, что тот ведет себя правильно и соблюдает основные правила спецификации байт-кода Java. Надежный компилятор Java не станет генерировать такой байт-код, который нарушает эти

условия. Тем не менее злоумышленник может намеренно собрать некорректный байт-код. Верификатор должен обнаружить эту опасность.

После того как код будет проверен, он считается безопасным в том смысле, что в нем нет случайных или злонамеренных ошибок определенных типов. Например, проверенный байт-код не может фальсифицировать ссылки или нарушать разрешения доступа к объектам (как в нашем примере с кредиткой). Он не может выполнять недопустимые преобразования типов и использовать объекты непредвиденным образом. Он даже не может порождать некоторые виды внутренних ошибок, таких как выход за верхнюю или нижнюю границу внутреннего стека. Эти фундаментальные гарантии лежат в основе всей модели безопасности Java.

Вы можете спросить: разве аналогичных мер безопасности нет во многих интерпретируемых языках? Конечно, вам не удастся повредить интерпретатор языка BASIC некорректной строкой написанного на BASIC кода, но не забывайте, что в большинстве интерпретируемых языков защита осуществляется на высоком уровне. В таких языках обычно есть ресурсоемкие интерпретаторы, выполняющие сложную работу на стадии выполнения, и поэтому эти языки неизбежно более медленные и громоздкие.

Для сравнения, байт-код Java представляет собой относительно простой низкоуровневый набор команд. И после того, как байт-код прошел предварительную статическую проверку, он затем выполняется интерпретатором на полной скорости и совершенно безопасно, без затратных проверок на стадии выполнения. Это решение стало одним из принципиально важных нововведений Java.

Верификатор является разновидностью математической «системы доказательства теорем». Проходя по байт-коду Java, верификатор анализирует его, применяя простые индуктивные правила. Такой анализ возможен, потому что скомпилированный байт-код Java содержит больше информации о типах, чем объектный код других похожих языков. Байт-код также должен соблюдать определенные правила, упрощающие его логику работы. Во-первых, многие команды байт-кода работают только с конкретными типами данных. Например, при операциях со стеком используются разные команды для обращения к объектам и к каждому из числовых типов Java. Аналогичным образом для каждого типа данных есть отдельные команды, перемещающие значение в локальную переменную и из нее.

Во-вторых, тип объекта, полученного в результате любой операции, всегда известен заранее. Никакие операции байт-кода не уничтожают значения переменных и не генерируют на выходе более одного типа значения. Поэтому всегда возможно проанализировать следующую команду и ее операнды, чтобы узнать тип получаемого значения.

Так как операция всегда производит заранее известный тип, типы всех элементов в стеке и локальных переменных в любой момент будущего могут быть определены по исходному состоянию. Совокупность всей информации о типах в любой конкретный момент называется *состоянием типов* стека; именно его Java анализирует перед запуском приложения. Java ничего не знает о фактических значениях, хранящихся в стеке и в переменных; известны только их типы. Однако этой информации достаточно для соблюдения правил безопасности и для уверенности в том, что с объектами не будут выполняться некорректные операции.

Чтобы иметь возможность анализировать состояние типов стека, Java устанавливает дополнительное ограничение на выполнение команд своего байт-кода: все пути к любой точке в коде должны завершаться с одним и тем же состоянием типов.

Загрузчики классов

Второй уровень безопасности Java — это *загрузчик классов*. Он отвечает за передачу байт-кода классов Java интерпретатору. Каждое приложение, которое загружает классы из сети, должно использовать для этого загрузчик классов.

После того как класс будет загружен и пройдет проверку верификатора, он остается связанным со своим загрузчиком классов. В результате классы эффективно делятся на пространства имен в зависимости от своего происхождения. Когда загруженный класс ссылается на другое имя класса, местонахождение нового класса предоставляется исходным загрузчиком классов. Это означает, что всем классам, полученным из конкретного источника, может быть разрешено взаимодействовать только с другими классами из того же источника. Например, браузер с поддержкой Java может использовать загрузчик классов, чтобы сформировать отдельное пространство для всех классов, загруженных по заданному веб-адресу (URL). Загрузчики классов также позволяют реализовать комплексную защиту на основе криптографических цифровых подписей классов.

Поиск классов всегда начинается со встроенных системных классов Java. Они загружаются из тех мест, которые указаны в переменной `classpath` интерпретатора Java (см. главу 3). Классы из `classpath` загружаются системой только один раз и остаются неизменными. Это означает, что приложение не может заменить фундаментальные системные классы собственными версиями с измененной функциональностью.

Менеджеры безопасности

Менеджер безопасности отвечает за принятие решений безопасности на уровне приложения. Он представляет собой объект, который может быть установлен

приложением для ограничения доступа к системным ресурсам. Менеджер безопасности получает возможность вмешиваться в происходящее каждый раз, когда приложение пытается обратиться к таким ресурсам, как файловая система, сетевые порты, внешние процессы и оконная среда. Менеджер безопасности может разрешить или отклонить каждый такой запрос.

Менеджеры безопасности в первую очередь представляют интерес для приложений, в которых выполнение ненадежного кода является частью нормальной работы. Например, браузер с поддержкой Java может запускать апплеты, загружая их из ненадежных источников в интернете. Одним из первых действий такого браузера должна стать установка менеджера безопасности. С этого момента менеджер безопасности будет ограничивать соответствующие виды доступа. У приложения появится возможность установить оптимальный уровень доверия перед выполнением произвольного блока кода. После того как менеджер безопасности будет установлен, его уже нельзя заменить.

Менеджер безопасности работает в сочетании с *контроллером доступа*, который позволяет реализовать политики безопасности на высоком уровне посредством редактирования файла декларативной политики безопасности. Политики доступа могут быть настолько простыми или сложными, насколько того требует конкретное приложение.

Иногда бывает достаточно запретить доступ ко всем ресурсам или к целым категориям сервисов (например, к файловой системе или к сети). Также возможно принятие сложных решений на основании высокоуровневой информации. Например, браузер с поддержкой Java может использовать такую политику доступа, которая позволяет пользователям указать уровень доверия для апплета и разрешать или запрещать доступ к определенным ресурсам в каждой конкретной ситуации. Конечно, предполагается, что браузер может определить, каким апплетам следует доверять. Вскоре вы узнаете, как эта проблема решается посредством цифровых подписей кода.

Целостность менеджера безопасности зависит от защиты, предоставляемой более низкими уровнями модели безопасности Java. Без гарантий, предоставляемых верификатором и загрузчиком классов, высокоуровневые предположения относительно безопасности системных ресурсов не имеют смысла. Безопасность, предоставляемая верификатором байт-кода Java, означает, что интерпретатор не может быть поврежден или фальсифицирован, а код Java будет использовать компоненты именно так, как следует. В свою очередь, это означает, что загрузчик классов может гарантировать, что приложение использует фундаментальные системные классы Java, а все обращения к базовым системным ресурсам могут осуществляться только через эти классы. При этих ограничениях контроль над такими ресурсами может быть централизован на высоком уровне с помощью менеджера безопасности и политики, определяемой пользователем.

Безопасность на уровнях приложения и пользователя

Есть тонкая граница между силой, достаточной для решения полезных задач, и силой делать все что угодно. Java закладывает основу для формирования безопасной среды, в которой ненадежный код может подвергаться карантину и выполняться безопасно. Тем не менее если вас не устраивает, что ваш код ограничивается узкими рамками «черного ящика» и выполняется в полной изоляции, вам придется разрешить ему доступ хотя бы к некоторым системным ресурсам. Каждый вид доступа сопряжен как с определенными рисками, так и с преимуществами. Например, в среде браузера преимущество предоставления ненадежному (неизвестному) апплету доступа к вашей оконной системе заключается в том, что он сможет отображать информацию. Значит, вы сможете выводить на экран что-то полезное. Но есть риск, что вместо этого апплет будет выводить какие-то бесполезные, надоедливые или оскорбительные слова.

Вот один крайний случай: даже простой запуск приложения предоставляет ему ресурс — машинное время. Оно может использоваться рационально или растрачиваться понапрасну. Трудно помешать ненадежному приложению расходовать ваше время или даже попытаться провести атаку «отказа в обслуживании». Другой крайний случай: мощное приложение, пользующееся доверием, может обоснованно требовать доступа к самым разным системным ресурсам (к файловой системе, к созданию процессов, к сетевым интерфейсам), а вредоносное приложение с доступом к этим же ресурсам может устроить хаос. Суть в том, что всегда необходимо уделять внимание важным проблемам безопасности, иногда очень сложным.

В некоторых ситуациях допустимо просто предложить пользователю подтвердить запрос кнопкой «ОК». Язык Java предоставляет средства для реализации любых политик безопасности. Тем не менее содержание этих политик в конечном счете зависит от вашей уверенности в происхождении и целостности кода. Здесь в игру вступают цифровые подписи.

Цифровые подписи в сочетании с сертификатами — это средства для проверки того, что данные действительно поступили из указанного источника и их никто не подменил по пути. Если банк снабжает своей цифровой подписью свое приложение «Чековая книжка», то вы можете убедиться в том, что приложение действительно получено от этого банка (а не от какого-то самозванца) и попало к вам в неизменном виде. Следовательно, вы можете разрешить своему браузеру доверять апплетам с цифровой подписью этого банка.

История Java

Язык активно развивается, и бывает трудно уследить, что в нем доступно сейчас, что обещали разработчики и что изменилось за последнее время. В последующих разделах приведен краткий рассказ о прошлом, настоящем и будущем Java. Не огорчайтесь, если какие-то термины будут непонятны. Некоторые из них мы рассмотрим в книге, а остальные вы можете самостоятельно изучить по мере появления практических навыков и уверенного владения основами Java. Что касается версий Java, в прилагаемых к ним файлах документации (release notes) от Oracle есть хорошие списки вносимых изменений со ссылками на более подробные сведения. Если вы используете старые версии, попробуйте почитать документацию Oracle (<https://docs.oracle.com/en/java>).

Прошлое: Java 1.0 — Java 13

Версия Java 1.0 предоставила базовую инфраструктуру для разработки Java: сам язык и пакеты, позволяющие писать апплеты и простые приложения. Версия 1.0 официально считается устаревшей, но все еще существует немало апплетов, соответствующих ее API.

Версия Java 1.1 заменила 1.0. В ней были реализованы значительные улучшения пакета AWT (Abstract Window Toolkit) — исходного средства разработки графических интерфейсов в Java, а также новые паттерны событий, новые языковые средства (такие, как рефлексия и внутренние классы) и другие крайне важные возможности. Эта версия много лет поддерживалась многими версиями браузеров Netscape и Internet Explorer. По разным причинам мир браузеров надолго застрял в этом состоянии.

Версия Java 1.2, получившая от Sun дополнительное название «Java 2», была выпущена в качестве основной версии в декабре 1998 года. Она содержала много усовершенствований и дополнений, прежде всего в наборе функций API, включенном в стандартные поставки. Самым заметным дополнением стал пакет разработки графических интерфейсов Swing, представленный в качестве фундаментального API, и новый полноценный API для двумерной графики. Swing — это усовершенствованный GUI-инструментарий для Java, заметно превосходящий по своим возможностям старый пакет AWT. (Swing, AWT и некоторые другие пакеты получили общее название JFC, или Java Foundation Classes). В версии 1.2 также появился полноценный API для работы с коллекциями.

Версия Java 1.3, выпущенная в начале 2000 года, включала ряд второстепенных улучшений, но в первую очередь была направлена на быстрое действие. В вер-

сии 1.3 код Java стал заметно быстрее работать на многих платформах, а заодно оказались исправленными многие ошибки пакета Swing. Тогда же происходило становление нескольких API корпоративного уровня, таких как Servlets и Enterprise JavaBeans.

В версию Java 1.4, выпущенную в 2002 году, был интегрирован новый набор API и многие давно ожидавшиеся возможности. В их число входили проверочные утверждения, регулярные выражения, API конфигураций и протоколирования, новая система ввода-вывода для крупномасштабных приложений, стандартная поддержка XML, фундаментальные усовершенствования в AWT и в Swing, а также развитая поддержка API сервлетов Java для веб-приложений.

Версия Java 5, выпущенная в 2004 году, стала основной версией, в которую были включены многие давно ожидавшиеся улучшения синтаксиса, в том числе: обобщения, безопасные по типам перечисления, расширенный цикл `for`, переменные списки аргументов, статическое импортирование, автоматическая упаковка и распаковка примитивных типов, а также усовершенствованные метаданные классов. Новый API многопоточности предоставил мощные средства управления потоками; также были добавлены API для форматирования печати и парсинга (разбора данных), аналогичные тем, которые есть в языке C. Механизм RMI (Remote Method Invocation) был переработан, чтобы исключить необходимость в скомпилированных каркасах и заглушках. Также был существенно дополнен стандартный XML API.

Версия Java 6, выпущенная в конце 2006 года, была относительно второстепенной. Она не добавила в язык Java никаких новых синтаксических элементов, зато содержала новые API расширений (например, для XML и веб-служб).

Версия Java 7, выпущенная в 2011 году, стала относительно существенным обновлением. В язык был внесен ряд дополнений — например, возможность использования строк в командах `switch` (об этом мы расскажем далее). Также за пять лет, прошедших с выхода Java 6, появились такие важные усовершенствования, как новая библиотека ввода-вывода `java.nio`.

В версии Java 8, выпущенной в 2014 году, была завершена работа над такими средствами, как лямбда-выражения и методы по умолчанию. (Ранее их исключили из Java 7 из-за того, что не успевали выпустить ее вовремя.) Также в восьмой версии была доработана поддержка даты и времени, в том числе возможность создания неизменяемых объектов с датами, удобных для использования в только что появившихся лямбда-выражениях.

В версии Java 9, выпущенной после нескольких задержек в 2017 году, появилась система модулей (Project Jigsaw), а также REPL-оболочка (Read-Evaluate-Print Loop) `jshell`. Мы будем использовать `jshell` во многих кратких примерах кода в последующих главах книги. В Java 9 из JDK также была исключена поддержка JavaDB.

Версия Java 10, выпущенная вскоре после Java 9 в начале 2018 года, получила обновленный механизм уборки мусора, а также ряд других возможностей, например корневые сертификаты для сборок OpenJDK. Была добавлена поддержка неизменяемых коллекций, а поддержка пакетов со старым стилем оформления (например, Aqua от Apple) прекратилась.

В версии Java 11, выпущенной в конце 2018 года, появился стандартный клиент HTTP и протокол TLS 1.3. Модули JavaFX и Java EE были удалены. (Модуль JavaFX был переработан в автономную библиотеку.) Также были исключены апплеты. Наряду с Java 8 версия Java 11 стала частью системы долгосрочной поддержки Oracle (LTS, Long Time Support). Некоторые версии — Java 8, Java 11 и, возможно, Java 17 — будут сопровождаться еще долго. Oracle пытается изменить процесс перехода пользователей и разработчиков на новые версии, однако у многих есть веские причины для того, чтобы продолжать пользоваться хорошо известными версиями. С планами и мыслями Oracle относительно LTS-версий и не-LTS-версий можно ознакомиться в Oracle Technology Network, в разделе «Oracle Java SE Support Roadmap» (<https://www.oracle.com/java/technologies/java-se-support-roadmap.html>).

В версии Java 12, выпущенной в начале 2019 года, были добавлены некоторые улучшения синтаксиса, в том числе предварительный вариант выражений `switch`.

Версия Java 13, выпущенная в сентябре 2019 года, включала предварительные варианты новых возможностей языка (например, текстовых блоков), а также значительное изменение реализации API сокетов. Согласно официальной документации, эта впечатляющая разработка предоставляет «более простую и более современную реализацию, которая упрощает сопровождение и отладку».

Настоящее: Java 14

В книгу включены все самые новые и полезные усовершенствования на момент последней фазы выпуска Java 14 весной 2020 года. В этой версии добавлен ряд улучшений синтаксиса в предварительных реализациях, обновлен механизм уборки мусора, удален API Pack200 и связанные с ним инструменты. Выражения `switch`, впервые представленные в Java 12, перешли из предварительного состояния в стандартный язык. К тому времени, когда вы будете читать эту книгу, появятся новые версии JDK, поскольку они выпускаются каждые полгода. Oracle хочет, чтобы разработчики рассматривали новые версии как обычные обновления функциональности. Для целей этой книги вам будет достаточно Java 11 — надежной версии с долгосрочной поддержкой. При изучении языка вам не надо беспокоиться обо всех его обновлениях, но если вы используете Java в реальных проектах, сверьтесь с «дорожной картой» Java, чтобы решить, имеет ли смысл идти ногу со временем. В главе 13 показано, как можно отслеживать

эту «дорожную карту» и как перерабатывать существующий код для использования новейших функций.

Сводка функциональности

Краткая сводка важнейших функциональных возможностей текущего базового API языка Java:

- *JDBC (Java Database Connectivity)* — основное средство для взаимодействия с базами данных (начиная с Java 1.1).
- *RMI (Remote Method Invocation)* — система распределенных объектов Java. RMI позволяет вызывать методы объектов, размещенных на сервере в другом месте сети (начиная с Java 1.1).
- *Java Security* — механизм управления доступом к системным ресурсам, объединенный с единым интерфейсом к криптографическим средствам. Java Security закладывает основу для классов с цифровыми подписями, о чем говорилось ранее.
- *Java Desktop* — объединяющий термин для множества функций, появившихся в Java 9, среди которых: компоненты пользовательского интерфейса Swing; «вариативный пользовательский интерфейс» (способность интерфейса адаптироваться к разным платформам); перетаскивание; 2D-графика; печать на принтерах; работа с изображениями и звуком; средства доступности (интеграция со специальными программами и с оборудованием для людей с ограниченными возможностями).
- *Интернационализация* — возможность написания программ, которые адаптируются к языку и локальному контексту, выбранному пользователем; программа автоматически выводит текст на подходящем языке (начиная с Java 1.1).
- *JNDI (Java Naming and Directory Interface)* — общая служба для просмотра ресурсов. JNDI объединяет доступ к службам каталогов, включая LDAP, Novell NDS и ряд других.

Далее перечислены API стандартных расширений языка. Некоторые из них (например, предназначенные для работы с XML и веб-сервисами) входят в стандартное издание Java; другие надо загружать отдельно и развертывать в вашем приложении или на сервере.

- *JavaMail* — унифицированный API для приложений, работающих с электронной почтой.
- *Java Media Framework* — еще один обобщающий термин, включающий Java 2D, Java 3D, Java Media Framework (фреймворк для координации вывода со многими разными типами контента), Java Speech (для распознава-

ния и синтеза речи), Java Sound (для работы со звуком высокого качества), Java TV (для интерактивного телевидения и аналогичных приложений) и т. д.

- *Сервлеты Java* — средство для создания веб-приложений, работающих на стороне сервера.
- *Криптография Java* — актуальные реализации криптографических алгоритмов. (Этот пакет был отделен от Java Security по юридическим причинам.)
- *XML / XSL* — средства для создания документов XML, для их обработки, проверки, отображения на объекты Java и обратно, преобразования при помощи таблиц стилей.

В этой книге мы стараемся дать вам представление о некоторых из этих возможностей. К сожалению для нас (но к счастью для разработчиков), среда Java теперь стала настолько богатой, что рассказать обо всем в одной книге уже невозможно.

Будущее

В наши дни язык Java уже не является модной новинкой, но остается одной из популярнейших платформ для разработки приложений. Это особенно справедливо в таких областях, как веб-службы, фреймворки веб-приложений и инструменты для работы с XML. Хотя языку Java не удалось доминировать на мобильных платформах (как ему, казалось бы, было суждено), тем не менее Java и его основные API можно использовать в программировании для мобильной операционной системы Google Android, используемой на миллиардах устройств по всему миру. В лагере Microsoft язык C#, производный от Java, захватил большую часть разработки .NET и принес базовый синтаксис и паттерны Java на соответствующие платформы.

Виртуальная машина Java (JVM) сама по себе является интересной областью исследования и развития. Появляются новые языки, которые используют набор возможностей и повсеместную доступность JVM. Clojure — мощный функциональный язык, у которого растет число поклонников в самых разных кругах. Kotlin — другой язык, убедительно завоевывающий популярность в сфере разработки для Android (где ранее господствовал Java). Это язык общего назначения, получающий широкое распространение в новых средах и функционально совместимый с Java.

Пожалуй, самые интересные области изменений в Java в наши дни связаны с двумя тенденциями: использование облегченных и упрощенных фреймворков для бизнеса, а также интеграция платформы Java с динамическими языками для сценарного программирования веб-страниц и расширений. Всех нас ждет еще более интересная работа.

Доступные средства

У вас есть выбор из нескольких вариантов сред разработки и исполнительных систем Java. Пакет Oracle JDK доступен для macOS, Windows и Linux. За дополнительной информацией о том, где получить новейшую версию JDK, обращайтесь на веб-сайт Java корпорации Oracle (<https://www.oracle.com/java/technologies>).

С 2017 года Oracle официально поддерживает обновления проекта с открытым кодом OpenJDK. Эта бесплатная версия может оказаться достаточной для отдельных программистов и для малых (и даже средних) компаний. Выпуски OpenJDK отстают от коммерческих выпусков и не включают профессиональную поддержку от Oracle, но Oracle твердо заявляет о сохранении бесплатного и открытого доступа к Java. Все примеры в книге мы писали и тестировали с помощью OpenJDK. Более подробную информацию вы найдете в первоисточнике, то есть на сайте OpenJDK (<https://openjdk.java.net>).

Для быстрой установки бесплатной версии Java 11 Amazon предоставляет свой дистрибутив Corretto с удобными инсталляторами для всех трех основных платформ. Версия Java 11 достаточна для всех примеров из этой книги, хотя мы упомянем и несколько возможностей из более поздних версий.

Также существует целый ряд интегрированных сред разработки (Integrated Development Environment, IDE) для Java. Одна из них рассматривается в этой книге: IntelliJ IDEA от компании JetBrains (<https://www.jetbrains.com/idea>) в бесплатном издании Community Edition. В этой многофункциональной среде разработки, содержащей полный набор необходимых инструментов, вам будет удобно писать, тестировать и упаковывать программы.

Первое приложение

Прежде чем браться за детальное рассмотрение языка Java, попробуем себя в деле: возьмем фрагменты работоспособного кода и немного поэкспериментируем с ними. В этой главе мы напишем маленькое приложение, которое демонстрирует многие концепции, встречающиеся в книге. Оно послужит своего рода презентацией основных возможностей языка и написанных на нем приложений.

Эта глава также служит кратким введением в объектно-ориентированные и многопоточные аспекты Java. Вероятно, эти концепции вам еще неизвестны, и в таком случае мы надеемся, что первое знакомство с ними в Java будет простым и приятным. А если вы уже работали в других объектно-ориентированных или многопоточных средах программирования, то наверняка оцените простоту и элегантность Java. Эта глава дает самый краткий обзор языка и общее представление о том, как им пользоваться. Не беспокойтесь, если какие-то из описанных концепций покажутся непонятными: мы подробнее расскажем о них в последующих главах.

Трудно переоценить важность экспериментов при изучении новых концепций — как в этой главе, так и в других. Не ограничивайтесь чтением примеров кода — выполняйте их. При каждой возможности мы будем показывать, как использовать `jshell` (подробнее см. раздел «Первые эксперименты с Java», с. 98), чтобы проверять фрагменты кода в реальном времени. Исходный код этих примеров, а также всех остальных примеров книги вы можете загрузить с портала GitHub (<https://github.com/10y/learnjava5e>). Компилируйте программы и проверяйте их в работе. А потом превращайте наши примеры в свои: экспериментируйте с ними, изменяйте их логику работы, ломайте, чините... в общем, получайте удовольствие.

Инструменты и среда Java

Теоретически для написания, компиляции и запуска простых Java-приложений достаточно пакета с открытым кодом Java Development Kit от Oracle (OpenJDK)

и обычного текстового редактора (vi, «Блокнот» и т. д.). Но на практике почти каждый современный программист пишет приложения в интегрированной среде разработки (IDE). Это дает много преимуществ: удобный просмотр исходного кода с цветовым выделением синтаксиса, помощь с навигацией, управление версиями исходного кода, встроенная документация, сборка, рефакторинг (переработка кода) и развертывание приложений — все эти возможности находятся прямо под рукой. По этой причине мы пропустим академическое описание работы с командной строкой и начнем с популярной бесплатной IDE, которая называется IntelliJ IDEA CE (Community Edition). Впрочем, если вам не хочется работать в IDE, используйте командную строку. Примеры команд: `javac>HelloJava.java` (для компиляции) и `java>HelloJava` (для запуска).

Для работы в среде IntelliJ IDEA надо сначала установить сам язык Java. В книге рассматривается версия Java 11 (с несколькими упоминаниями нововведений в версиях 12 и 13). Хотя примеры кода из этой главы будут успешно работать и в более ранних версиях, лучше установить JDK версии 11 или выше, чтобы все примеры из книги гарантированно компилировались. JDK содержит некоторые средства разработчика, которые будут рассмотрены в главе 3. Чтобы узнать, какая версия установлена на вашем компьютере (и установлена ли), введите команду `java -version` в командной строке. Если Java отсутствует или версия более ранняя, чем JDK 11, загрузите нужную версию с сайта OpenJDK (<https://jdk.java.net>). Вы найдете там весь ряд версий для Linux, macOS и Windows (<https://jdk.java.net/archive>).

Интегрированную среду IntelliJ IDEA вы можете загрузить с сайта компании JetBrains (<https://www.jetbrains.com/idea/download>). Для работы с этой книгой, как и для создания множества приложений, более чем достаточно бесплатного издания Community. Загружаемый файл представляет собой инсталлятор .exe (или архив .zip) для Windows, инсталлятор .dmg для macOS или архив .tar.gz для Linux. При необходимости распакуйте архив и запустите инсталлятор. В конце книги, в приложении (с. 497), приведена более подробная информация о загрузке и установке IDEA, а также о том, как загрузить примеры кода из книги.

Установка JDK

Следует сразу сказать, что вы можете свободно загружать и использовать официальный коммерческий пакет Oracle JDK для личных целей. На сайте Oracle (<https://www.oracle.com/java>) всегда есть новейшая версия, а также ряд предшествующих версий, в том числе с долгосрочной поддержкой. Старые версии иногда бывают нужны разработчикам в целях совместимости.

Но если вы планируете использовать Java в коммерческих целях или в составе проектной группы, то для таких случаев Oracle JDK предоставляется на строгих условиях платного лицензирования. Из-за этого (и по другим, более философ-

ским причинам) мы обычно используем бесплатный пакет OpenJDK, упоминавшийся ранее. К сожалению, в этой версии с открытым кодом нет удобных программ установки (инсталляторов) для разных платформ. Если вам нужна простота установки и версия с долгосрочной поддержкой (например, Java 8 или 11), выберите другой дистрибутив OpenJDK, например Amazon Corretto (<https://aws.amazon.com/ru/corretto>).

Для читателей, которые хотят иметь свободу выбора версии Java и не боятся небольшой работы по настройке, мы расскажем, как устанавливать OpenJDK на каждой из трех основных платформ. Для примера мы выбрали версию Java 13.0.1 — последнюю на момент написания книги. Независимо от того, в какой операционной системе вы работаете, для загрузки OpenJDK перейдите на соответствующую страницу на сайте Oracle (<https://jdk.java.net/archive>).

Установка OpenJDK в Linux

Файл, загружаемый для типичных систем Linux, представляет собой TAR-архив (`.tar.gz`). Вы можете распаковать его в любой общий каталог по вашему выбору (например, `/usr/lib/jvm`). Запустите приложение терминала и выполните следующие команды¹, чтобы перейти в каталог загрузки (например, `Downloads`), распаковать архив и проверить Java:

```
~ $ cd Downloads

~/Downloads $ sudo tar xvf openjdk-13.0.1_linux-x64_bin.tar.gz \
--directory /usr/lib/jvm
...
jdk-13.0.1/lib/src.zip
jdk-13.0.1/lib/tzdb.dat
jdk-13.0.1/release

~/Downloads $ /usr/lib/jvm/jdk-13.0.1/bin/java -version
openjdk version "13.0.1" 2019-10-15
OpenJDK Runtime Environment (build 13.0.1+9)
OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
```

После успешной установки Java настройте терминал для использования этой среды, присвоив значения переменных `JAVA_HOME` и `PATH`. Чтобы убедиться в правильности этой конфигурации, проверьте версию компилятора Java `javac`:

¹ Длинные команды, которые не умецаются в длину одной строки книги, условно показаны в виде нескольких строк с разделителями `\` в конце, обозначающими продолжение команды на следующей строке. Все вводимые команды, независимо от их длины, выделены жирным шрифтом.

```
~/Downloads $ cd
~ $ export JAVA_HOME=/usr/lib/jvm/jdk-13.0.1
~ $ export PATH=$PATH:$JAVA_HOME/bin
~ $ javac -version
javac 13.0.1
```

Изменения в `JAVA_HOME` и `PATH` надо закрепить, обновив стартовые сценарии или сценарии `rc` для вашей оболочки. Например, обе строки `export`, только что использованные в терминале, можно добавить в файл `.bashrc`.

Также надо заметить, что многие дистрибутивы Linux предоставляют доступ к некоторым версиям Java через свои менеджеры пакетов. Поищите в интернете информацию по строкам вида «install java ubuntu» или «install java redhat» и посмотрите, есть ли для вашей системы альтернативные способы установки Java, которые лучше соответствуют вашему привычному стилю работы в Linux.

Установка OpenJDK в macOS

Для пользователей macOS установка OpenJDK похожа на установку в Linux: загрузите архив `.tar.gz` и распакуйте его в нужное место. В отличие от Linux, «нужное место» определяется вполне конкретно¹.

Воспользуйтесь приложением «Терминал» (Terminal) из папки **Applications ▸ Utilities**, чтобы распаковать и переместить папку с OpenJDK:

```
~ $ cd Downloads
Downloads $ tar xf openjdk-13.0.1_osx-x64_bin.tar.gz
Downloads $ sudo mv jdk-13.0.1.jdk /Library/Java/JavaVirtualMachines/
```

Команда `sudo` позволяет пользователям с административными привилегиями выполнять специальные действия, обычно зарезервированные для «суперпользователя». Вам будет предложено ввести пароль. После перемещения папки JDK задайте значение переменной среды `JAVA_HOME`. Команда `java`, включенная в macOS, теперь сможет найти установленную версию.

¹ Но это не обязательно, если вы опытный пользователь *nix и знаете, как работать с переменными среды и путями. В таком случае вы можете распаковать архив в любой удобный каталог. Возможно, вам понадобится «научить» другие приложения использовать Java из этого каталога, так как многие приложения ограничиваются поиском по «хорошо известным» каталогам.


```
Downloads $ cd ~  
  
~ $ export \  
    JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-13.0.1.jdk/Contents/Home  
  
~ $ java -version  
openjdk version "13.0.1" 2019-10-15  
OpenJDK Runtime Environment (build 13.0.1+9)  
OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
```

Как и в случае с Linux, вам надо добавить строку `JAVA_HOME` в соответствующий стартовый файл (например, в файл `.bash_profile` в вашем домашнем каталоге), если вы будете работать с Java в командной строке.

У пользователей macOS 10.15 (Catalina) и более поздних версий могут возникнуть некоторые сложности при установке и проверке Java. Вследствие изменений в macOS корпорация Oracle еще не сертифицировала Java для Catalina (на момент выхода книги). Конечно, Java все равно можно запускать в системах Catalina, но наиболее сложные приложения могут столкнуться с ошибками. Заинтересованные пользователи могут прочитать технические заметки Oracle по использованию JDK с Catalina. В первой части этих заметок рассматривается установка официального JDK, а вторая часть посвящена установке из архива `.tar.gz`, описанной выше.

Установка OpenJDK в Windows

Системы Windows разделяют многие концепции с системами *nix, хотя пользовательские интерфейсы для работы с этими концепциями сильно различаются. Загрузите архив OpenJDK для Windows — это должен быть файл `.zip` (вместо файла `.tar.gz`). Распакуйте файл в нужную папку. Как и в случае с Linux, «нужную папку» выбираете вы. Мы создали папку `Java` в `C:\Program Files` и поместили в нее содержимое архива, как показано на рис. 2.1.

Когда папка JDK появится на своем месте, настройте переменные среды (по аналогии с macOS и Linux). Самый быстрый способ получить доступ к переменным среды — провести поиск по строке «environment» («переменные среды») и найти в результатах поиска вариант «Edit the system environment variables» («Изменение системных переменных среды»), как показано на рис. 2.2.

Сначала надо создать новую запись для переменной `JAVA_HOME` и дополнить строку `Path` информацией о Java. Мы решили добавить эти изменения в системную часть (System variables), но если вы единственный пользователь своего компьютера, их также можно добавить в пользовательскую часть (User variables).

Создайте новую переменную `JAVA_HOME` и присвойте ей значение: путь к папке, в которой установлен JDK (рис. 2.3).



Рис. 2.1. Папка Java в Windows

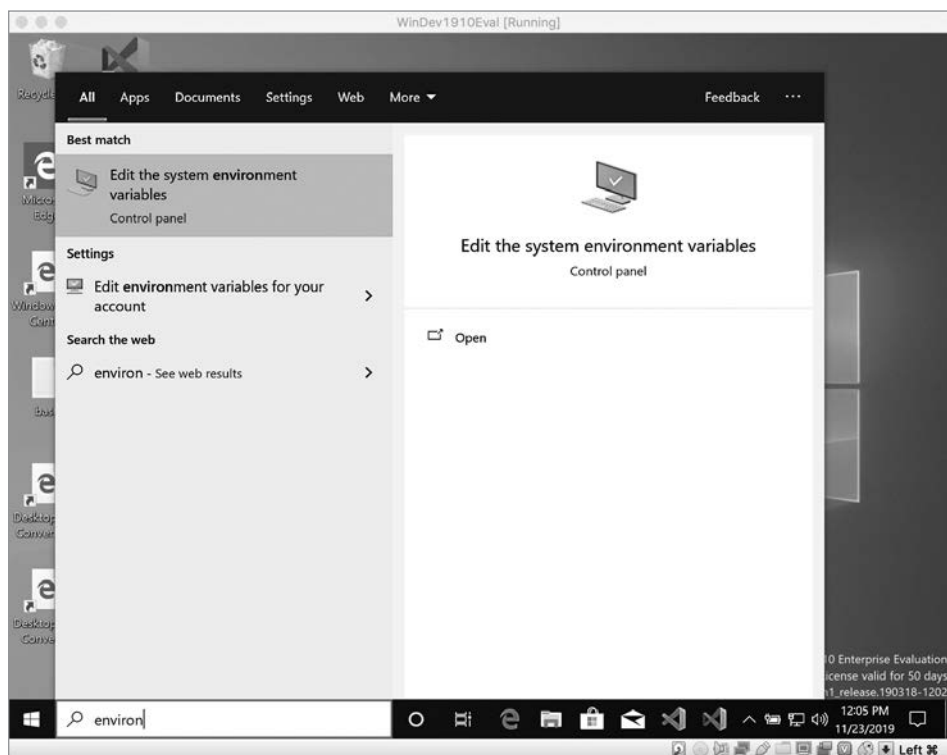


Рис. 2.2. Поиск редактора переменных среды в Windows

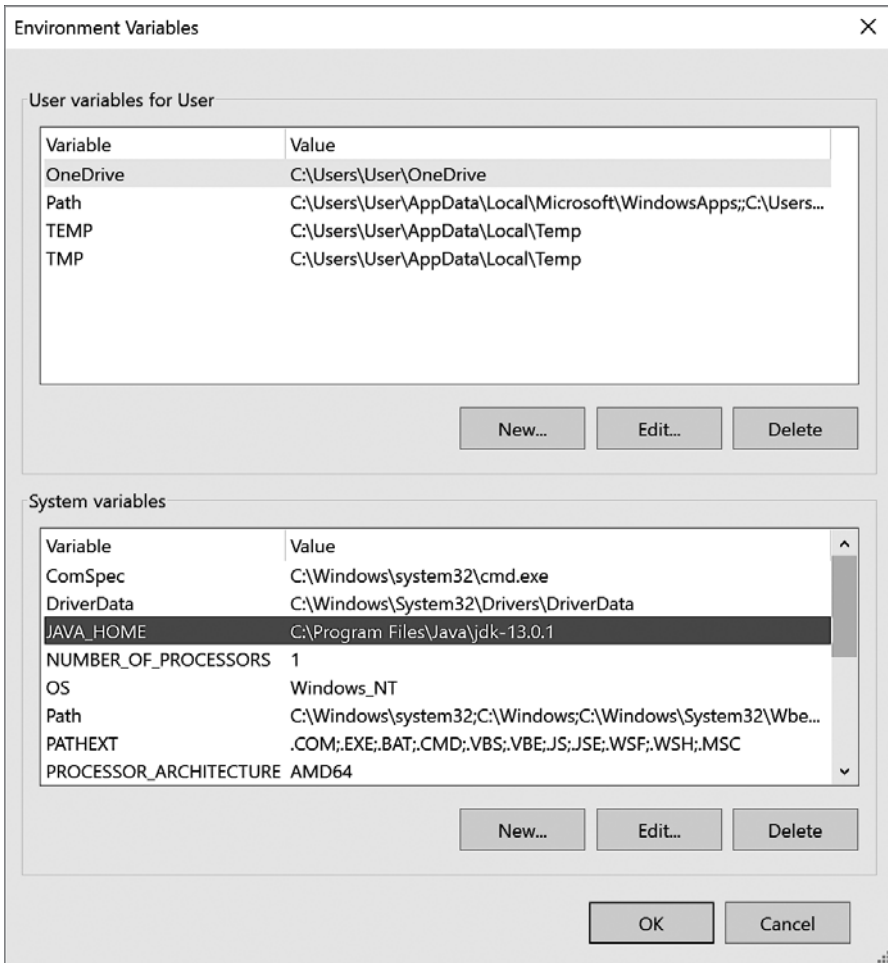


Рис. 2.3. Создание переменной среды `JAVA_HOME` в Windows

После того как вы присвоите значение переменной `JAVA_HOME`, добавьте соответствующую запись в переменную `Path`, чтобы Windows знала, где искать программы `java` и `javac`. Эта запись должна указывать на папку `bin`, находящуюся в папке с Java. Чтобы указать в `Path` значение переменной `JAVA_HOME`, заключите ее имя между символами `%` (`%JAVA_HOME%`), как показано на рис. 2.4.

Приложение «Командная строка» (Command Prompt) выполняет в Windows те же функции, что и терминалы в macOS и Linux. Запустите это приложение и введите команду для проверки версии Java. Результат выглядит примерно так, как показано на рис. 2.5.

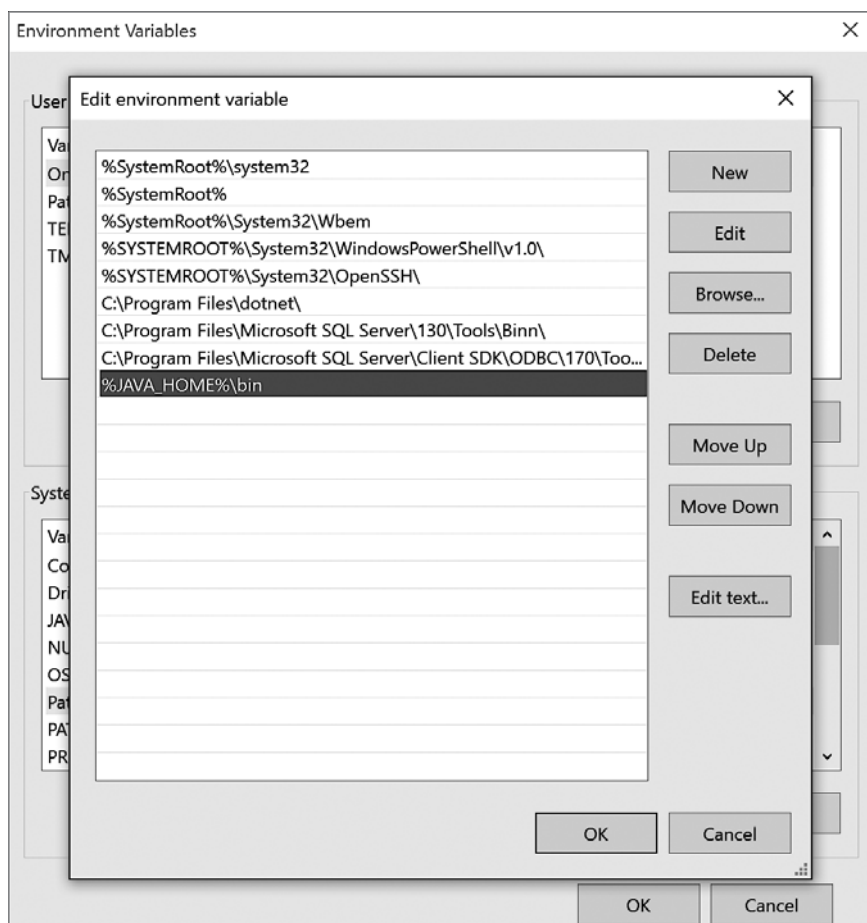


Рис. 2.4. Редактирование переменной среды Path в Windows

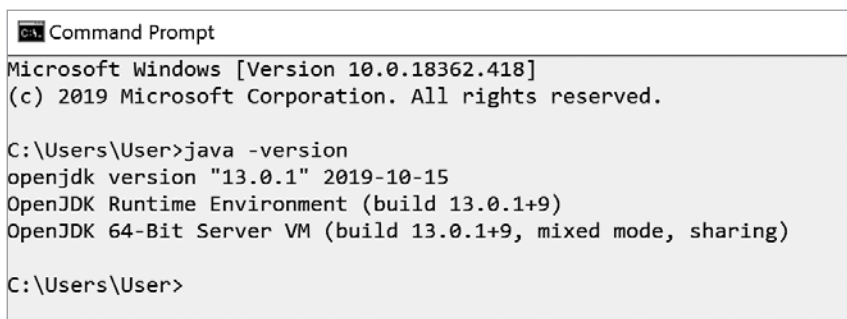


Рис. 2.5. Проверка версии Java в Windows

Конечно, вы можете и дальше работать с Java в командной строке, но лучше указать в настройках IntelliJ IDEA путь к папке с JDK, а затем постоянно использовать эту удобную среду разработки.

Настройка конфигурации IntelliJ IDEA и создание проекта

При первом запуске IDEA выберите создание нового проекта («New Project»). Затем убедитесь, что в строке «Project SDK» указана версия Java 11 или выше, как показано на рис. 2.6, и нажмите кнопку Next.



Рис. 2.6. Первое диалоговое окно нового проекта

Теперь выберите шаблон `Command Line App`. Он содержит минимальный класс Java с методом `main()`, который можно выполнить. (В последующих главах будет подробно рассмотрена структура программ Java и команд, из которых они состоят.) Выбрав шаблон так, как показано на рис. 2.7, нажмите кнопку `Next`.

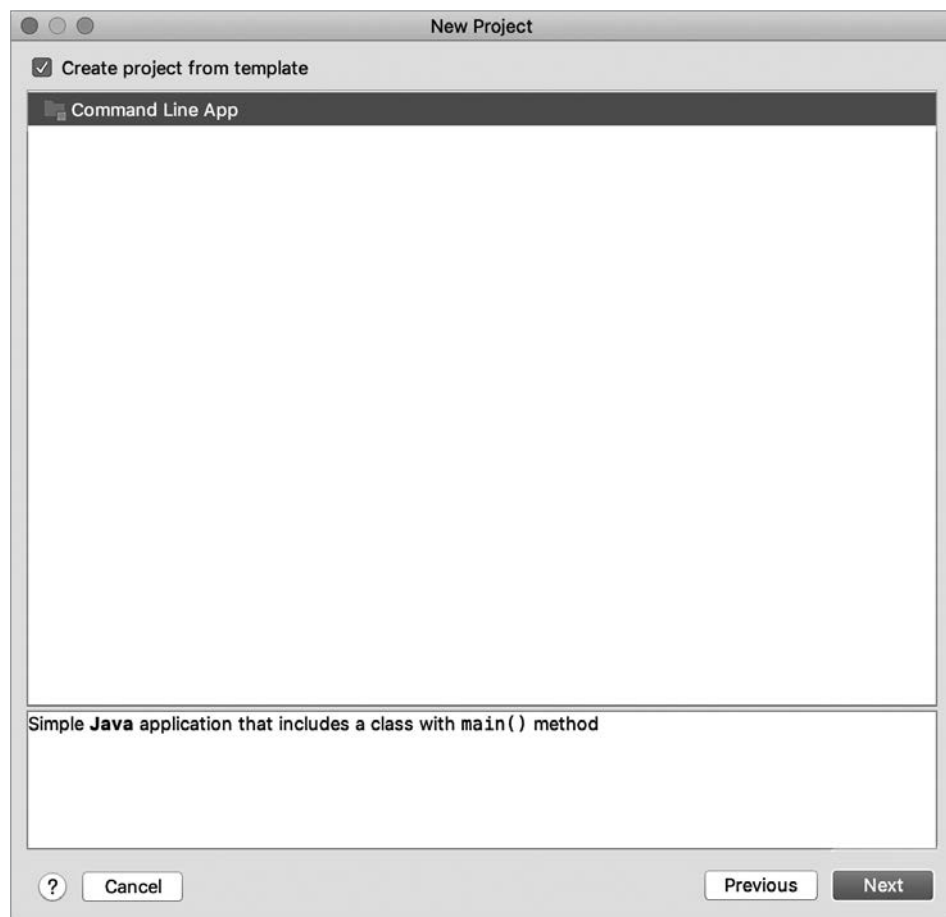


Рис. 2.7. Выбор шаблона для проекта

Наконец, введите имя проекта. Для учебного приложения из этой главы мы выбрали имя `HelloJava`. IDEA предложит путь к папке для файлов этого проекта; она будет внутри заданной по умолчанию папки для всех проектов IDEA. В других случаях вы могли бы нажать кнопку с многоточием (...), чтобы выбрать

любую другую папку на компьютере. Когда эти два поля будут заполнены, нажмите кнопку **Finish** (рис. 2.8).

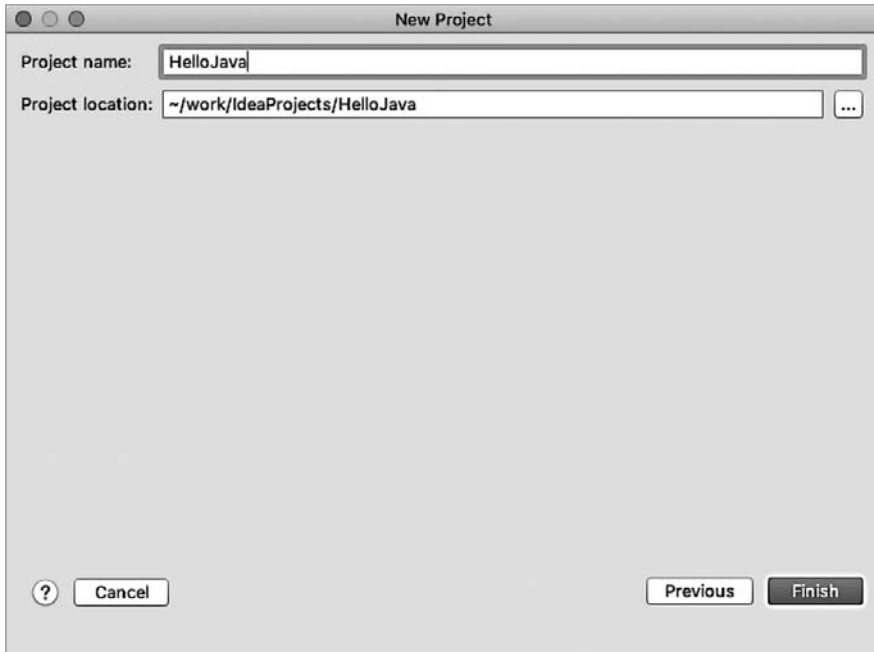


Рис. 2.8. Выбор имени проекта и папки для его файлов

Поздравляем! Вы создали программу Java... почти. В нее надо добавить строку кода, которая будет выводить сообщение на экран.

Добавьте следующую строку между фигурными скобками, после строки `public static void main(String[] args):`

```
System.out.println("Hello, World!");
```

Готовая программа должна выглядеть примерно так, как в правой части окна на рис. 2.9.

Сначала мы запустим этот код, а затем слегка дополним его, чтобы он стал чуть менее тривиальным. В последующих главах начнутся более интересные примеры, в которых мы будем использовать все больше и больше элементов Java. Тем не менее процедура создания всех примеров будет более или менее похожей. Запомните эти подготовительные действия, они вам еще не раз пригодятся.

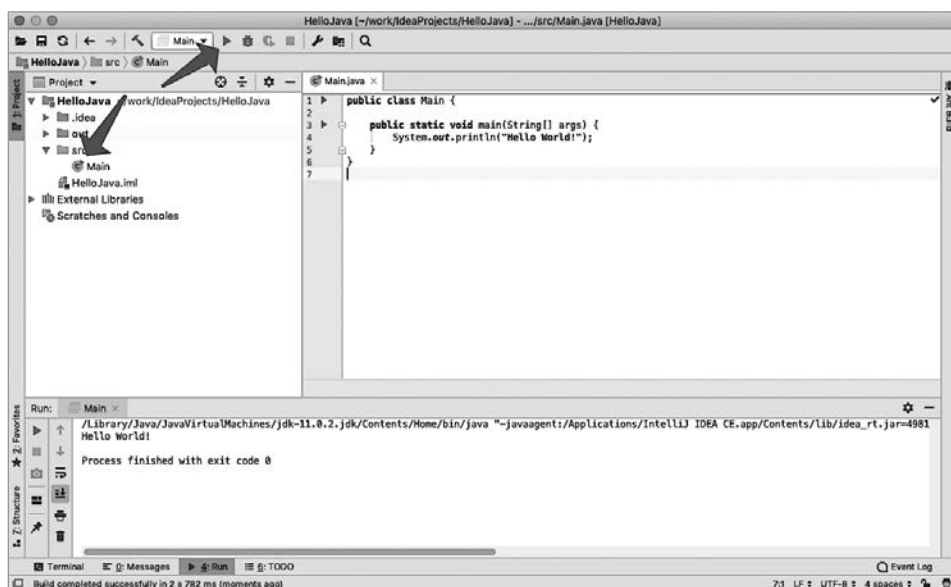


Рис. 2.9. Запуск программы

Запуск программы

Простой шаблон, который вам предоставила IDEA, стал хорошей отправной точкой для вашей первой программы. Обратите внимание: на верхней панели инструментов вы видите название класса `Main`, а рядом находится кнопка запуска программы, обозначенная зеленой стрелкой (см. рис. 2.9). Эта кнопка означает, что IDEA понимает, как запустить метод `main()` в этом классе. Попробуйте нажать эту кнопку. На вкладке `Run` в нижней части окна появится сообщение «Hello World!». И снова поздравляем: вы только что запустили свою первую программу на Java.

Загрузка примеров кода

Все примеры кода, приведенные в книге, мы разместили на сайте GitHub. Он считается основным облачным репозиторием как для общедоступных проектов с открытым кодом, так и для коммерческих проектов с закрытым кодом. Кроме простых средств для хранения исходного кода и контроля версий, GitHub предоставляет много полезных инструментов. Если вы будете писать приложение или библиотеку, которыми захотите поделиться с другими, зарегистрируйтесь на GitHub и хорошо изучите его возможности. Но ZIP-файлы общедоступных проектов можно загружать без регистрации, как показано на рис. 2.10.

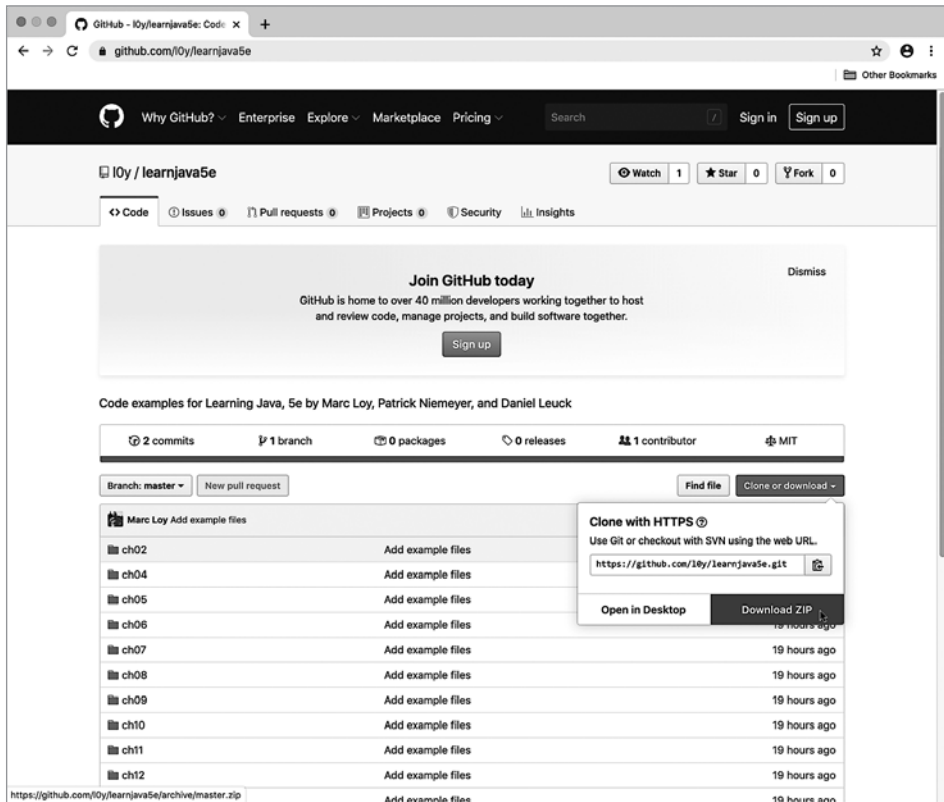


Рис. 2.10. Загрузка ZIP-архива с GitHub

В итоге у вас должен быть загружен файл с именем `learnjava5e-master.zip`; это архив «главной» (master) ветви репозитория. Если вы знакомы с GitHub по другим проектам, то можете клонировать репозиторий, но это не обязательно: наш ZIP-файл содержит все необходимое для запуска примеров кода, приведенных в книге.

После распаковки загруженного файла будут созданы папки для всех глав, содержащих примеры, а также папка `game` с кодом простой игры. Эта игра демонстрирует в одном завершенном приложении большую часть концепций программирования, представленных в книге. И примеры, и игра подробно рассматриваются в следующих главах.

Как упоминалось ранее, любой пример кода, взятый из этого ZIP-файла, можно скомпилировать и запустить прямо из командной строки. Также вы можете импортировать код в свою любимую IDE. В приложении (с. 497) подробно рассказано о том, как лучше всего импортировать эти примеры в IntelliJ IDEA.

HelloJava

По давней программистской традиции мы предлагаем начать изучение языка с простейшего приложения в стиле «Hello World», которое мы назвали `HelloJava`.

Затем, прежде чем расстаться с этим примером, мы несколько раз его доработаем (`HelloJava`, `HelloJava2`, `HelloJava3`), дополняя новыми возможностями и демонстрируя новые концепции. Начнем с минимальной версии:

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

Эта программа из пяти строк объявляет класс с именем `HelloJava` и метод с именем `main()`. В ней используется стандартный (заранее определенный в языке) метод `println()`, выводящий строку текста. Это *программа для командной строки*, то есть она выполняется в командном интерпретаторе, напоминающем «окно DOS», и выводит там свои результаты. Ранее вы использовали шаблон от IDEA, в котором у класса было имя `Main`. В этом нет ничего неправильного, но при создании более сложных программ лучше давать классам более содержательные имена, что мы и постараемся делать в примерах кода. Независимо от имени класса, вывод в командную строку считается немного старомодным, поэтому прежде чем двигаться дальше, мы создадим для `HelloJava` графический интерфейс (GUI). Пока не отвлекайтесь на код; просто продолжайте читать, а мы вскоре вернемся к объяснениям.

Вместо строки, вызывающей метод `println()`, мы создадим объект `JFrame` для размещения окна на экране. Для начала заменим строку, содержащую `println`, следующими тремя строками:

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

Этот фрагмент кода создает объект `JFrame` с заголовком «Hello, Java!». Объект `JFrame` представляет собой графическое окно. Чтобы оно появилось на экране, мы просто указываем его размер с помощью метода `setSize()`, а затем делаем его видимым с помощью метода `setVisible()`.

Если бы мы на этом остановились, то на экране появилось бы пустое окно с текстом «Hello, Java!» в заголовке. Но мы выведем сообщение и в самом окне, не ограничиваясь его заголовком. Для этого понадобится еще пара строк. В следующем примере добавлен объект `JLabel`, который выводит надпись, выровненную по центру окна. Первая строка с оператором `import` сообщает Java, где искать

классы `JFrame` и `JLabel` (то есть определения используемых в программе объектов `JFrame` и `JLabel`).

```
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 );
    }
}
```

Чтобы скомпилировать и запустить этот код, выберите класс `ch02/HelloJava.java` на панели проекта в левой части окна IDEA, а затем снова нажмите кнопку **Run** на панели инструментов (кнопку с зеленой стрелкой). Результат показан на рис. 2.11.

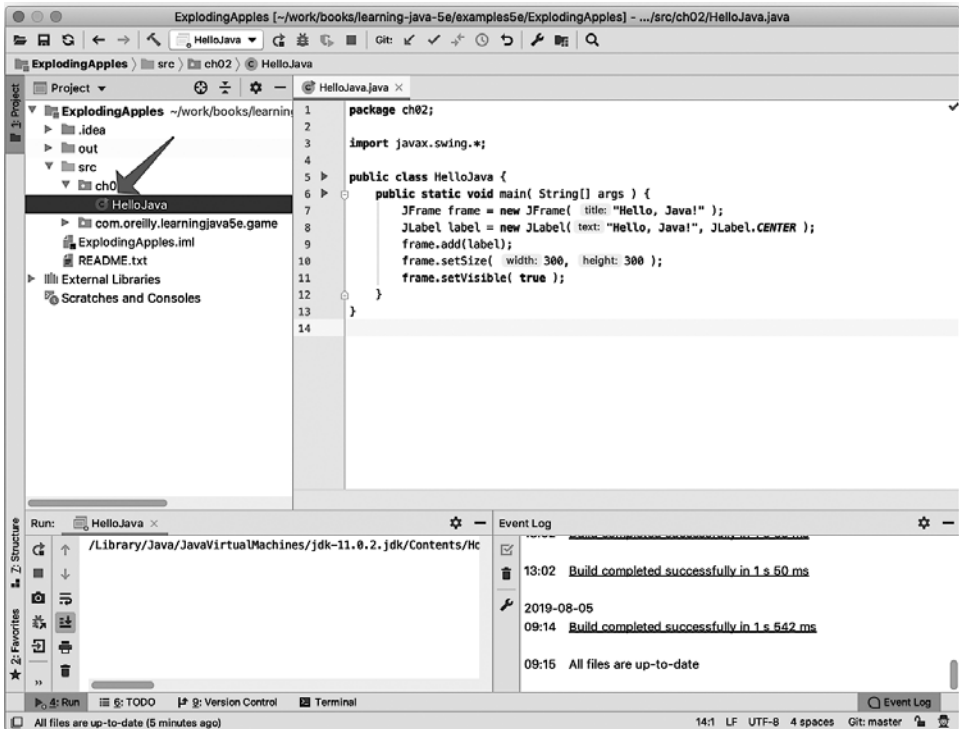


Рис. 2.11. Запуск приложения HelloJava

Вы должны увидеть на экране примерно такое окно, как на рис. 2.12. Очередные поздравления, вы запустили свое второе приложение на Java! Погрейтесь в лучах славы от экрана своего монитора.

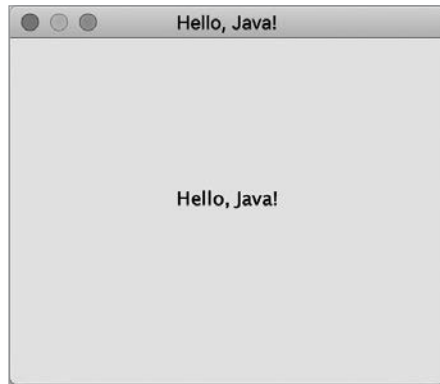


Рис. 2.12. Работающее приложение HelloJava

Учтите, что при щелчке на кнопке закрытия окна оно исчезает с экрана, но программа продолжает работать. (Этот недостаток мы исправим в последующих версиях.) Чтобы завершить программу в IDEA, нажмите кнопку с красным квадратом; она находится рядом с кнопкой запуска (зеленой стрелкой). А если программа запущена в командной строке, нажмите для завершения клавиши `Ctrl+C`. Кстати, ничто не мешает вам запустить сразу несколько экземпляров (копий) программы.

Программа `HelloJava` очень мала, но в ней происходит много всего, что не заметно с первого взгляда. Эти несколько строк кода — всего лишь вершина айсберга, а под ней скрываются несколько уровней функциональности, предоставляемой языком Java и его библиотеками Swing. В этой главе мы быстро пройдем немалый путь, чтобы показать вам общую картину. Мы постараемся привести достаточно информации, чтобы вы хорошо понимали, что происходит в каждом примере, но отложим подробные объяснения до последующих глав. Это относится и к синтаксическим элементам языка Java, и к соответствующим объектно-ориентированным концепциям программирования. Итак, давайте разберемся, что же происходит в нашем первом примере.

Классы

Прежде всего определяется класс с именем `HelloJava`:

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

Классы — это основные структурные элементы Java и большинства других объектно-ориентированных языков. *Класс* — это группа элементов данных и связанных с ними функций, способных выполнять операции с этими данными. Элементы данных класса называются *переменными* (или иногда *полями*), а функции называются в Java *методами*. Основные достоинства объектно-ориентированных языков — это взаимосвязь между данными и функциональностью в классах, а также способность классов к *инкапсуляции*, то есть к сокрытию деталей их внутреннего устройства. Инкапсуляция позволяет разработчику не беспокоиться об этих низкоуровневых деталях.

В приложении класс может представлять нечто конкретное: кнопку на экране, информацию в электронной таблице и т. д. Класс может представлять и что-то более абстрактное: скажем, алгоритм сортировки списка или чувство уныния у персонажа видеоигры. Например, класс, представляющий электронную таблицу, может содержать переменные, хранящие значения отдельных ячеек, и методы для выполнения операций с этими ячейками («очистить строку», «вычислить значения» и т. д.).

Класс `HelloJava` представляет собой целое приложение Java, написанное в виде одного класса. Он определяет всего один метод `main()`, содержащий все тело нашей программы:

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

При запуске приложения метод `main()` вызывается первым. Фрагмент `String[] args` нужен, чтобы передавать приложению *аргументы командной строки*. Мы подробно разберем метод `main()` в следующем разделе. Наконец, следует заметить, что эта версия нашего приложения не определяет никакие переменные как части своего класса, но использует две переменные, `frame` и `label`, внутри своего метода `main()`. Вскоре мы поговорим о переменных более подробно.

Метод `main()`

Как было показано ранее, при запуске Java-приложения выбирается конкретный класс, имя которого передается в качестве аргумента виртуальной машине Java. Когда мы это сделали, команда `java` проверила класс `HelloJava` и выяснила, есть ли в нем специальный метод с именем `main()`, соответствующий определенной форме. Такой метод есть, поэтому он был выполнен. Если бы его не было, мы бы получили сообщение об ошибке. Метод `main()` является точкой входа для всех приложений. Каждое автономное приложение Java содержит как минимум один класс с методом `main()`, который выполняет необходимые действия для запуска всего остального, что есть в приложении.

Метод `main()` создает окно (`JFrame`), в которое направляется экранный вывод класса `HelloJava`. В данном случае этот метод выполняет всю работу приложения. Но обычно в объектно-ориентированных приложениях обязанности делегируются многим разным классам. Так мы сделаем в следующей версии: добавим второй класс и покажем, что в ходе эволюции нашего приложения метод `main()` останется более или менее неизменным — он просто содержит стартовую процедуру инициализации.

Кратко разберем метод `main()`, чтобы пояснить, что он делает. Сначала `main()` создает объект класса `JFrame` — обычное окно с заголовком:

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

Слово `new` в этой строке кода крайне важно. `JFrame` — это имя класса, формирующего различные окна на экране. Но сам этот класс представляет собой только шаблон — что-то вроде «чертежа», по которому строятся все объекты такого типа. Ключевое слово `new` приказывает Java выделить память и создать в памяти один конкретный объект класса `JFrame`. В данном случае аргумент в круглых скобках сообщает этому объекту, какой текст надо вывести в заголовке. Мы могли бы убрать слова “Hello, Java” и оставить пустые круглые скобки, чтобы получить `JFrame` без текста в заголовке (но только потому, что такая возможность специально предусмотрена в классе `JFrame`).

Все окна `JFrame` в момент создания имеют очень малый размер. Прежде чем отображать окно на экране, назначим ему ширину и высоту:

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

Эта строка — пример вызова метода для конкретного объекта. В данном случае метод `setSize()`, определенный в классе `JFrame`, работает с конкретным объектом класса `JFrame`. Этот объект присвоен переменной `frame`. Затем мы аналогичным образом создаем экземпляр типа `JLabel`, представляющий собой надпись:

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

Объект класса `JLabel` можно сравнить с листом бумаги. Он содержит текст, расположенный в определенном месте, — как и в нашем окне. Это объектно-ориентированная концепция: вы используете объект для того, чтобы **хранить** в нем текст, вместо того чтобы «написать» текст вызовом соответствующего метода и двигаться дальше. Скоро вы поймете, почему это делается именно так.

Теперь разместим надпись в только что созданном окне:

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

Здесь для размещения надписи `label` внутри объекта класса `JFrame` используется метод `add()`. Объект класса `JFrame` является своего рода контейнером, в котором

могут находиться другие объекты. Вскоре мы поясним эту концепцию. Последняя задача для метода `main()` — отобразить на экране окно и его содержимое, которое иначе осталось бы невидимым. Согласитесь, что приложение с невидимым окном было бы довольно скучным.

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

Вот и весь метод `main()`. Рассматривая другие примеры этой главы, вы увидите, что по мере изменения класса `HelloJava` этот метод останется в основном неизменным.

Классы и объекты

Класс — это своего рода «чертеж» для какого-либо компонента приложения; в классе содержатся те методы и переменные, из которых состоит этот компонент. Когда приложение работает, в нем могут существовать несколько работающих копий класса. Эти отдельные «воплощения» класса называются его *экземплярами* или *объектами*. Два экземпляра конкретного класса могут содержать разные данные, но методы у них всегда одинаковые.

Для примера возьмем стандартный класс кнопки `Button`. Существует только один класс `Button`, но приложение может создать много разных объектов класса `Button`; все они будут экземплярами одного и того же класса. Более того, два экземпляра `Button` могут содержать разные данные, то есть кнопки могут по-разному выглядеть и выполнять разные действия. В этом смысле класс можно рассматривать как шаблон для изготовления представляемых им объектов. Класс — это что-то вроде «формочки для печенья», которой мы «штампует» готовые к работе экземпляры (объекты) в памяти компьютера. Как будет показано далее, это еще не все (классы могут определять информацию, которая будет совместно использоваться всеми экземплярами), но пока достаточно и этого краткого описания. Понятия классов и объектов детально объясняются в главе 5.

Термин «объект» очень многозначен и иногда, в некоторых контекстах, используется почти как синоним термина «класс». Объекты — это абстрактные сущности, которые в той или иной форме встречаются во всех объектно-ориентированных языках. Мы будем использовать термин «объект» для обозначения экземпляра класса. Таким образом, экземпляр класса `Button` может называться кнопкой, объектом `Button` или просто объектом.

Метод `main()` в предыдущем примере создает единственный экземпляр класса `JLabel` и выводит его внутри экземпляра класса `JFrame`. При желании попробуйте изменить метод `main()` таким образом, чтобы он создал несколько экземпляров `JLabel`, отображаемых в одном окне или в нескольких разных окнах.

Переменные и типы

В Java каждый класс определяет собственный *тип* (*тип данных*). Вы можете создать переменную этого типа, а затем сохранить в ней экземпляр этого класса. Например, переменная может иметь тип `Button` и хранить экземпляр класса `Button`, или иметь тип `SpreadSheetCell` и хранить объект `SpreadSheetCell`, или иметь один из простых типов (таких, как `int` или `float`), представляющих числа. Все переменные обладают типами и не могут хранить произвольные (не соответствующие этим типам) объекты — это важное свойство языка Java, которое обеспечивает безопасность и корректность кода.

Если ненадолго забыть о переменных, используемых внутри метода `main()`, то в нашем примере `HelloJava` объявляется только одна переменная `args`. Это происходит в объявлении метода `main()`:

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

Подобно функциям в других языках программирования, метод в Java объявляет список параметров (переменных), которые должны передаваться ему в аргументах, с указанием типа каждого параметра. В нашем примере метод `main` требует, чтобы при вызове ему передавался один параметр: массив объектов `String` в переменной с именем `args`. `String` — фундаментальный класс для представления текста в Java. С помощью параметра `args` приложение получает аргументы командной строки, переданные при его запуске виртуальной машине Java. (В данном случае мы их не используем.)

До настоящего момента мы упрощенно говорили, что переменные могут хранить в себе объекты. На самом деле в переменных, имеющих типы классов, хранятся не сами объекты, а ссылки на них. *Ссылка* (*reference*) — это указатель на объект (то есть адрес в памяти, по которому находится объект). Если объявить переменную с типом какого-либо класса, не присвоив ей объект, она не будет указывать ни на что. По умолчанию ей будет присвоено значение `null`, которое означает «значение не определено». Если вы попытаетесь использовать переменную со значением `null` таким образом, как если бы она ссылалась на реальный объект, то в результате возникнет *ошибка времени выполнения* (*runtime error*) `NullPointerException`.

Конечно, ссылка на объект должна откуда-то появиться. В нашем примере два объекта были созданы оператором `new`. Создание объектов мы подробно рассмотрим далее в этой главе.

HelloComponent

До сих пор все приложение `HelloJava` состояло из одного класса. Более того, из-за простоты оно уместилось в единственном методе. И хотя мы использова-

ли пару объектов для вывода текста в окне, наш код пока не имеет объектно-ориентированной структуры. Сейчас мы это исправим, добавив в программу второй класс. Для пользы дела мы откажемся от стандартного класса `JLabel` (прости-прощай, `JLabel`!) и заменим его нашим собственным графическим классом `HelloComponent`. Сначала класс `HelloComponent` будет совсем простым: он будет выводить сообщение «Hello, Java!» в фиксированной позиции. Затем мы расширим его функциональность.

Код нового класса очень краток, мы написали всего 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 фрагмент в файл `HelloJava.java` или поместить в отдельный файл с именем `HelloComponent.java`. Если вы выбрали первый вариант, перенесите новую команду `import` в начало кода, вслед за другой командой `import`. Чтобы использовать новый класс вместо `JLabel`, просто замените две строки с упоминанием `label` следующей строкой:

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

Теперь при компиляции исходного кода `HelloJava.java` будут созданы два двоичных файла классов: `HelloJava.class` и `HelloComponent.class` (какой бы вариант организации исходного кода вы ни выбрали). Результат выполнения кода почти не отличается от версии с `JLabel`, но при изменении размеров окна вы заметите, что в нашем классе нет автоматического выравнивания надписи по центру.

Что же было сделано и почему мы приложили такие усилия, отказавшись от великолепно работающего компонента `JLabel`? Мы создали новый класс `HelloComponent`, расширив базовый графический класс с именем `JComponent`. *Расширение* класса — это создание нового класса посредством добавления функциональности в существующий класс. (Эта тема рассматривается в следующем разделе.) Итак, мы создали расширенную разновидность класса `JComponent` с методом `paintComponent()`, который отвечает за вывод сообщения. Наш метод `paintComponent()` получает один аргумент — переменную с именем `g` (пожалуй, слишком коротким), имеющую тип `Graphics`. При вызове метода `paintComponent()` переменной `g` присваивается объект `Graphics`, который используется в теле метода. Метод `paintComponent()` и класс `Graphics` будут рассмотрены далее. Вы поймете, зачем они понадобились, когда мы займемся добавлением новых функций в свой компонент.

Наследование

Классы Java образуют иерархию «родитель — потомок», в которой родитель называется *суперклассом*, а потомок — *субклассом*, или *подклассом*. Эти концепции будут рассмотрены в главе 5. В Java у каждого класса есть ровно один суперкласс (один родитель), но может быть много субклассов. Единственное исключение из правила составляет класс `Object`, находящийся на самой вершине иерархии: у него нет суперкласса.

В объявлении класса в предыдущем примере ключевое слово `extends` указывает, что наш `HelloComponent` расширяет класс `JComponent`, то есть является субклассом класса `JComponent`:

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

Субкласс наследует (полностью или частично) переменные и методы своего суперкласса. Благодаря наследованию, субкласс может использовать эти переменные и методы так, как если бы он сам объявил (определил) их. Субкласс может добавить к ним собственные переменные и методы. Субкласс также может переопределить или изменить унаследованные от суперкласса методы, и в этом случае каждый такой метод скрывается (замещается) своей новой версией, определенной в субклассе. Таким образом, наследование дает программисту мощный механизм, в котором каждый субкласс может уточнять и расширять функциональность своего суперкласса.

Например, на основе гипотетического класса, представляющего электронную таблицу, можно сделать субкласс **научной** электронной таблицы, содержащий дополнительные математические функции и встроенные константы. В таком случае исходный код научной электронной таблицы будет объявлять методы для дополнительных математических функций и переменные для констант, но при этом новый класс автоматически включит в себя все переменные и методы, образующие функциональность основной (более простой) электронной таблицы; они унаследуются от родительского класса. Также это означает, что «научная электронная таблица» не перестает быть «электронной таблицей», а расширенную версию можно использовать везде, где возможно использование основной, более простой. (Как будет показано далее, этот факт имеет очень глубокие и принципиальные последствия.) Иными словами, более специализированные объекты могут использоваться вместо более общих, а их поведение может настраиваться без изменения всего остального кода приложения. Этот принцип, называемый *полиморфизмом*, является одним из столпов объектно-ориентированного программирования.

Наш класс `HelloComponent` — это субкласс класса `JComponent`, то есть он наследует много переменных и методов, которые не объявлены явно в его исходном коде. Поэтому он может быть полноценным компонентом в составе `JFrame`, для этого нужны лишь незначительные настройки.

Класс JComponent

Класс `JComponent` предоставляет основу для создания всевозможных компонентов пользовательского интерфейса (GUI-компонентов). Все конкретные компоненты — кнопки, надписи, списки и т. д. — реализуются как subclasses `JComponent`.

Мы переопределяем методы в subclasses, чтобы реализовать нужную логику работы конкретного компонента. На первый взгляд кажется, что такой подход устанавливает излишние ограничения, так как мы получаем в свое распоряжение только заранее определенный набор функций. Но это впечатление обманчиво. Учтите, что методы, о которых идет речь, предназначены только для взаимодействия с оконной системой. В них не нужно втискивать весь код приложения. Реальные приложения состоят из сотен и тысяч классов, каждый из которых содержит множество собственных методов и переменных. Подавляющее большинство объектов в реальном приложении связано с его спецификой; они называются *объектами предметной области*. А класс `JComponent` и другие стандартные (предопределенные) классы Java служат основой только для тех небольших фрагментов кода, которые обрабатывают события пользовательского интерфейса и выводят информацию для пользователя.

Метод `paintComponent()` играет важную роль в классе `JComponent`; мы переопределяем этот метод, чтобы реализовать отображение нашего конкретного компонента на экране. По умолчанию `paintComponent()` не выводит ничего. Если бы мы не переопределили его в subclasses, то наш компонент остался бы невидимым. В данном случае мы переопределяем `paintComponent()`, чтобы он делал что-то более интересное. Никакие другие унаследованные методы и поля `JComponent` не переопределяются, потому что они предоставляют для нашего простого примера базовую функциональность и разумные значения по умолчанию. По мере роста нашего приложения `HelloJava` мы лучше изучим унаследованные методы и будем использовать дополнительные методы. Также мы добавим некоторые методы и переменные, предназначенные специально для `HelloComponent`.

`JComponent` в действительности является лишь вершиной огромного айсберга под названием `Swing`. `Swing` — это GUI-инструментарий Java, который мы включили в наш пример командой `import` в начале кода. Мы подробнее рассмотрим `Swing` в главе 10.

Отношения между классами

Класс `HelloComponent` можно рассматривать как `JComponent`, и это не будет ошибкой, потому что при subclassировании создается отношение «является частным случаем». Любой subclasses — это «частный случай» своего суперкласса, а также любого из вышестоящих суперклассов. Далее мы поближе познакомимся

с иерархией классов Java, а пока взгляните на рис. 2.13 — вы увидите, что класс `JComponent` является субклассом класса `Container`, который, в свою очередь, унаследован от `Component` и т. д.

В этом смысле наш `HelloComponent` — это «частный случай» класса `JComponent`, а также класса `Container`, и каждый из них может считаться «частным случаем» класса `Component`. От этих трех вышестоящих классов `HelloComponent` наследует свою базовую функциональность в пользовательском интерфейсе и (как будет показано далее) возможность встраивания в него других графических компонентов.

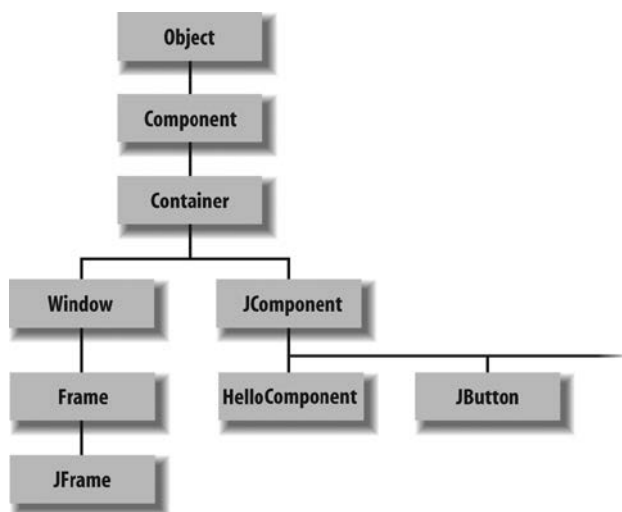


Рис. 2.13. Часть иерархии классов Java

`Component` является субклассом класса `Object` верхнего уровня. Любой класс является «частным случаем» `Object`. Все остальные классы Java API наследуют часть своей логики работы от класса `Object`, который определяет несколько базовых методов, как будет показано в главе 5. В международной терминологии словом «object» (с маленькой буквы) называют объект, то есть экземпляр любого класса, а именем `Object` (с большой буквы) обозначают конкретный суперкласс, самый верхний в иерархии Java.

Пакеты и импортирование

Ранее мы упоминали, что первая строка нашего примера сообщает Java, где следует искать некоторые из используемых классов:

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

Точнее говоря, она сообщает компилятору, что мы собираемся использовать классы из GUI-инструментария Swing (в нашем случае их три: `JFrame`, `JLabel` и `JComponent`). Эти классы объединены в *пакет* Java с именем `javax.swing`. Пакет Java — это группа классов, родственных по функциональности или области применения. Классы одного пакета имеют особые привилегии взаимного доступа и часто разрабатываются с расчетом на тесное взаимодействие.

Имена пакетов строятся по иерархическому принципу. Их компоненты перечисляются через точку, например `java.util` или `java.util.zip`. Классы пакета должны составлять по правилам, позволяющим находить их по путям в списке `classpath`. Классы получают имя пакета как часть своего полного, или *полностью уточненного, имени* (*fully qualified name*). Например, полное имя класса `JComponent` имеет вид `javax.swing.JComponent`. На класс можно ссылаться напрямую по полному имени, и в этом случае оператор `import` не требуется:

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

Строка `import javax.swing.*` в начале кода позволяет затем ссылаться на все классы пакета `javax.swing` по простым именам. Таким образом, нам не придется использовать полные имена для обозначения классов `JComponent`, `JLabel` и `JFrame`.

Как было показано при добавлении в код второго класса, исходный файл Java может содержать одну или несколько строк с оператором `import`. Эти строки создают «путь поиска», который сообщает компилятору Java, где искать классы, когда вы обращаетесь к ним по простым именам. (На самом деле это не путь, но он предотвращает появление неоднозначных имен, которые могут вызвать ошибки.) Мы использовали специальную запись `*` для обозначения того, что импортируется весь пакет. Но при необходимости можно импортировать не весь пакет, а только один класс. Например, в нашем примере из всего пакета `java.awt` используется только класс `Graphics`. Следовательно, вместо импортирования всех классов пакета AWT (Abstract Window Toolkit) символом `*` можно было бы ограничиться одним классом: `import java.awt.Graphics`. Впрочем, далее нам будут нужны и другие классы из пакета AWT.

Иерархии пакетов `java.` и `javax.` имеют особый смысл. Любой пакет, начинающийся с префикса `java.`, является частью базового Java API и доступен на любой платформе, поддерживающей Java. Имя пакета `javax.` обычно обозначает стандартное расширение базовой платформы, которое может быть установлено (или не установлено) в вашей системе. Однако за последние годы многие стандартные расширения были добавлены в базовый Java API без переименования. Примером служит пакет `javax.swing`; несмотря на свое имя, он входит в состав базового API. На рис. 2.14 показаны некоторые базовые пакеты Java с указанием одного-двух характерных классов.

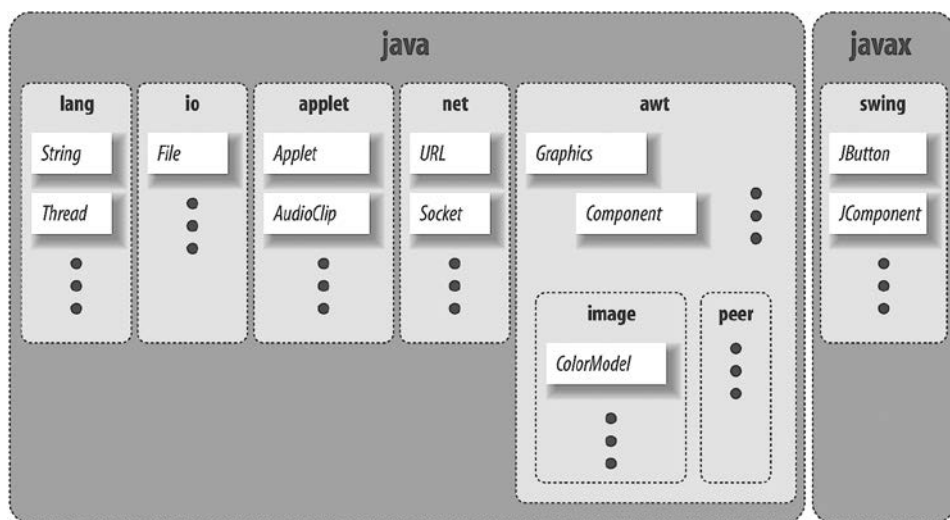


Рис. 2.14. Некоторые базовые пакеты Java

Пакет `java.lang` содержит фундаментальные классы, необходимые самому языку Java. Этот пакет импортируется автоматически, поэтому нам не потребовался оператор `import` для работы с такими классами, как `String` и `System`. Пакет `java.awt` содержит классы традиционного графического инструментария AWT; пакет `java.net` содержит классы для сетевых приложений и т. д.

По мере изучения Java вы будете все лучше понимать, как важно хорошо разбираться во всех доступных пакетах: что делает каждый из них, когда и как им пользоваться. Это абсолютно необходимо для того, чтобы стать успешным Java-разработчиком.

Метод `paintComponent()`

Исходный код класса `HelloComponent` определяет метод `paintComponent()`, переопределяющий метод `paintComponent()` класса `JComponent`:

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

Метод `paintComponent()` вызывается тогда, когда компонент должен вывести себя на экран. Он получает один аргумент (объект `Graphics`) и не возвращает вызывающей стороне никакого значения (`void`).

Модификаторы — это ключевые слова, которые ставятся перед именами классов, переменных и методов для изменения их доступности, логики работы или смысла. Метод `paintComponent()` объявлен с модификатором `public`. Это означает, что он может вызываться методами других классов, а не только класса `HelloComponent`. В данном случае метод `paintComponent()` **вызывается оконной средой Java**. А методы и переменные, объявленные с модификатором `private`, доступны только из того класса, которому они принадлежат.

Объект `Graphics`, то есть экземпляр класса `Graphics`, представляет конкретную область графического вывода (также называемую *графическим контекстом*). Он содержит методы, которые нужны для рисования в этой области, и переменные, представляющие такие характеристики, как режимы отсечения и рисования. Конкретный объект `Graphics`, передаваемый методу `paintComponent()`, соответствует той области экрана, которую `HelloComponent` занимает внутри `JFrame`.

Класс `Graphics` предоставляет методы для вывода геометрических фигур, изображений и текста. Метод `drawString()` объекта `Graphics` вызывается в классе `HelloComponent` для вывода сообщения в области с заданными координатами.

Как мы уже показали, чтобы вызвать метод какого-либо объекта, надо после имени объекта написать точку (.) и затем имя метода. Например, следующей строкой мы вызываем метод `drawString()` объекта `Graphics`, на который ссылается наша переменная `g`:

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

Возможно, вы не сразу привыкнете к мысли о том, что графическим выводом в нашем приложении занимается метод, который вызывается откуда-то извне (из оконной среды Java), да еще и в произвольные моменты времени. Как можно сделать что-то полезное в таких условиях? Как контролировать, что и когда происходит? На эти вопросы мы скоро ответим. А пока просто подумайте, какую структуру вы бы предложили для приложения, которое реагирует на команды вместо того, чтобы рисовать по собственной инициативе.

HelloJava2: продолжение

Разобравшись с азами, добавим в наше приложение чуть больше интерактивности. Следующее маленькое обновление позволит перетаскивать надпись мышью. Впрочем, начинающему программисту это обновление покажется не таким уж маленьким. Не бойтесь! Все темы, представленные в этом примере, будут подробнее изложены в следующих главах. А пока поэкспериментируйте с нашим очередным примером; он поможет вам увереннее создавать и запускать программы Java, даже если вы еще не до конца освоились с содержащимся в них кодом.

Мы назовем этот пример `HelloJava2`, чтобы не создавать лишней путаницы в попытках переделать прежнюю версию. Основные изменения — это расширение функциональности класса `HelloComponent` и его переименование с целью избежать конфликтов имен (`HelloComponent2`, `HelloComponent3`). Вы уже видели, как работает наследование, и у вас может появиться вопрос: не лучше ли создать субкласс класса `HelloComponent` и воспользоваться наследованием, чтобы написать новый пример на базе существующего и расширить его функциональность? Нет, в данном случае это не принесло бы пользы, а для наглядности мы просто напишем класс заново.

Приложение `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 перерисовать текст сообщения
        messageX = e.getX();
        messageY = e.getY();
        repaint();
    }

    public void mouseMoved( MouseEvent e ) { }
}
```


Две косые черты (//) означают, что вся последующая часть строки содержит комментарий. Мы написали в `HelloJava2` несколько комментариев, чтобы вам было проще следить за происходящим.

Сохраните текст этого примера в файле с именем `HelloJava2.java` и скомпилируйте его так же, как прежде. В результате должны быть созданы два новых файла классов: `HelloJava2.class` и `HelloComponent2.class`.

Запустите программу следующей командой:

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

Или, если вы вводите примеры в IDEA, нажмите кнопку Run. Замените сообщение «Hello, Java!» любым другим и наслаждайтесь, перетаскивая его по экрану. Обратите внимание: при нажатии кнопки закрытия окна приложение завершается; этот факт будет объяснен далее, когда мы будем обсуждать события. А теперь посмотрим, что же изменилось.

Переменные экземпляра

В класс `HelloComponent2` были добавлены некоторые переменные:

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

Переменные `messageX` и `messageY` — это целые числа, определяющие текущие координаты перемещаемой надписи. Мы инициализировали их значениями по умолчанию, с которыми надпись будет выводиться недалеко от центра окна. Целочисленные переменные в Java (`int`) представляют 32-разрядные числа со знаком, поэтому в них удобно хранить значения координат. Переменная `theMessage` относится к типу `String` и может хранить экземпляры класса `String`, то есть текст.

Обратите внимание: эти три переменные объявляются в фигурных скобках в определении класса, но не внутри какого-либо конкретного метода в этом классе. Такие переменные называются *переменными экземпляра* и принадлежат всему объекту в целом. Копии таких переменных присутствуют в каждом отдельном экземпляре класса. Переменные экземпляра всегда видны и могут использоваться методами внутри своего класса. В зависимости от своих модификаторов они также могут быть доступны за пределами класса.

Если переменные экземпляра не инициализируются явно, то по умолчанию им присваивается значение `0`, `false` или `null` в зависимости от типа переменной. Числовые типы инициализируются значением `0`, логические — значением `false`, а переменным с типом класса в таких случаях присваивается значение `null`,

означающее «нет никакого значения». Попытка использовать объект со значением `null` приводит к ошибке времени выполнения (`runtime error`).

Переменные экземпляра отличаются от аргументов методов и от других переменных, объявляемых в области видимости конкретного метода. Последние называются *локальными переменными*. Фактически это скрытые переменные, которые видны только своему методу (или своему блоку кода). Java не инициализирует локальные переменные, поэтому вам надо их явно инициализировать. При попытке обращения к локальной переменной, которой еще не присвоено значение, происходит ошибка компиляции (`compile-time error`). Локальные переменные продолжают существовать во время выполнения метода, а затем пропадают (но перед этим их значения можно где-то сохранить). При каждом вызове метода его локальные переменные создаются заново, и им должны быть присвоены значения.

Мы использовали новые переменные, чтобы сделать слишком банальный метод `paintComponent()` более динамичным. Теперь все аргументы вызова `drawString()` определяются этими переменными.

Конструкторы

Класс `HelloComponent2` включает особый метод, называемый *конструктором*. Конструктор вызывается для создания нового экземпляра класса. При создании нового объекта Java выделяет для него память, присваивает переменным экземпляра значения по умолчанию и вызывает метод — конструктор класса, выполняющий все необходимые подготовительные действия на уровне приложения.

Имя конструктора всегда совпадает с именем класса. Например, конструктор класса `HelloComponent2` называется `HelloComponent2()`. Конструктор не имеет возвращаемого значения, и вы можете исходить из того, что он просто создает объект с типом своего класса. Как и другие методы, конструкторы могут получать аргументы. Единственный смысл их существования — настройка и инициализация «новорожденных» экземпляров класса, иногда с использованием информации, переданной в параметрах.

Объект создается оператором `new` с указанием конструктора класса и всех необходимых аргументов. Полученный экземпляр объекта возвращается как значение. В нашем примере новый экземпляр `HelloComponent2` создается в методе `main()` следующей строкой:

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

На самом деле в этой строке выполняются две операции. Можно записать их в виде двух отдельных строк, чтобы они стали более понятными:

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

Особенно важна первая строка, в которой создается новый объект `HelloComponent2`. Конструктор `HelloComponent2` получает в аргументе объект `String` и, как мы запрограммировали, использует его для настройки сообщения, выводимого в окне. Благодаря небольшой помощи со стороны компилятора Java, заключенный в кавычки текст в исходном коде Java преобразуется в объект `String`. (Класс `String` рассматривается в главе 8.) Вторая строка просто добавляет наш новый компонент в окно, чтобы сделать его видимым, как это делалось в предыдущих примерах.

Если вы хотите, чтобы сообщение можно было задать при запуске программы, замените строку конструктора следующей:

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

Теперь текст сообщения можно передать в командной строке при запуске приложения следующей командой:

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

`args[0]` обозначает первый параметр командной строки. Смысл этой конструкции станет более понятным при рассмотрении массивов в главе 4. Если вы работаете в интегрированной среде разработки, то перед запуском приложения ее надо настроить, указав параметры командной строки, как показано для IntelliJ IDEA на рис. 2.15.

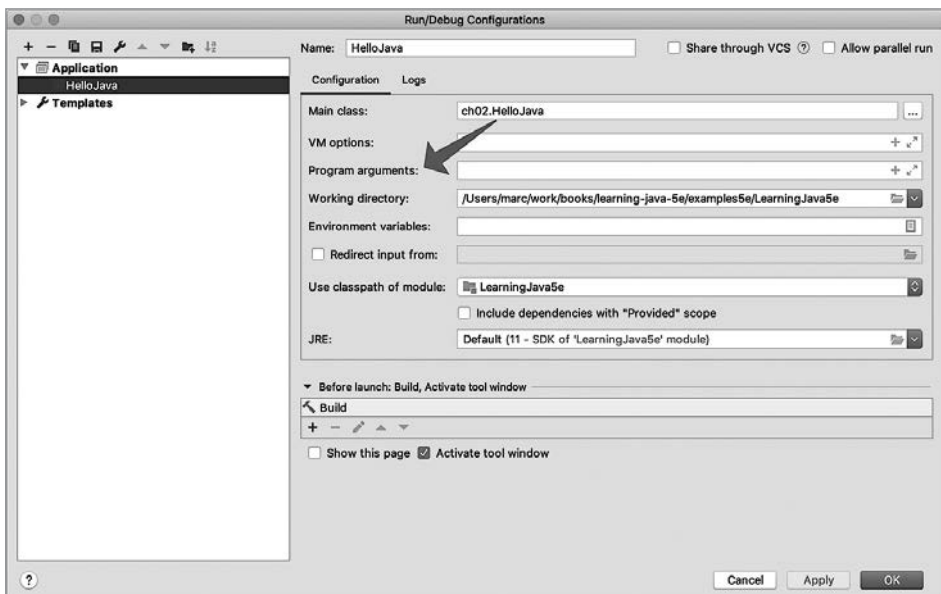


Рис. 2.15. Диалоговое окно IDEA для передачи параметров командной строки

Затем конструктор `HelloComponent2` решает две задачи: он задает текст переменной экземпляра `theMessage` и вызывает метод `addMouseMotionListener()`. Этот метод является частью механизма событий, о котором будет рассказано ниже. Он сообщает системе: «Меня интересует все, что происходит с мышью».

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

Специальная, доступная только для чтения переменная `this` используется для явных обращений к объекту (к контексту «этого» объекта) при вызове `addMouseMotionListener()`. Метод может использовать `this` для обращения к экземпляру объекта, которому он принадлежит. Таким образом, следующие команды эквивалентны — обе присваивают значение переменной экземпляра `theMessage`:

```
theMessage = message;
```

или:

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

Обычно для ссылок на переменные экземпляра используется более короткая, неявная форма, но ключевое слово `this` нам еще пригодится, когда потребуется явно передать ссылку на текущий объект методу из другого класса. Обычно это делается для того, чтобы методы других классов вызывали наши общедоступные методы или использовали открытые переменные.

События

В классе `HelloComponent2` есть еще два метода: `mouseDragged()` и `mouseMoved()`. Они нужны для получения информации от мыши. Каждый раз, когда пользователь выполняет некоторое действие (например, нажимает клавишу на клавиатуре, перемещает мышь или касается сенсорного экрана), Java генерирует *событие*. Событие представляет выполненное действие; оно содержит информацию о действии (например, его время и координаты). Большинство событий связано с конкретными GUI-компонентами в приложении. Например, нажатие клавиши может соответствовать вводу символа в конкретном текстовом поле, а щелчок кнопкой мыши может активизировать конкретную кнопку на экране. Даже простое перемещение указателя мыши в определенную область экрана может активизировать такие эффекты, как цветовое выделение или изменение формы указателя.

Чтобы работать с этими событиями, мы импортировали новый пакет `java.awt.event`. Он предоставляет определенные классы `Event`, необходимые для получения информации от пользователя. Обратите внимание: импорт пакета `java.awt.*` не приводит к автоматическому импортированию пакета `event`. Импорт не является рекурсивным. Пакеты не содержат другие пакеты, несмотря на то что иерархическая схема имен создает такое впечатление.

Существует много разных классов событий, включая `MouseEvent`, `KeyEvent` и `ActionEvent`. В основном их смысл вполне очевиден. Событие `MouseEvent` происходит тогда, когда пользователь что-то делает с мышью; событие `KeyEvent` происходит при нажатии клавиши и т. д. Событие `ActionEvent` немного особенное; вы увидите его в действии в главе 10. А пока сосредоточимся на обработке событий `MouseEvent`.

GUI-компоненты в Java генерируют события для конкретных видов действий пользователя. Например, если нажать кнопку мыши внутри компонента, этот компонент сгенерирует событие мыши. Объекты могут запрашивать получение событий от одного или нескольких компонентов, регистрируя *слушатель* для нужного источника событий. Например, чтобы объявить, что слушатель хочет получать события перемещения мыши, надо вызвать метод `addMouseMotionListener()` соответствующего компонента, передав этому методу объект слушателя в аргументе. Именно это происходит в конструкторе нашего класса: компонент вызывает собственный метод `addMouseMotionListener()` с аргументом `this`, что означает «Я хочу получать относящиеся ко мне события перемещения мыши».

Так мы регистрируемся для получения событий. Но как мы их получаем? Для этого используются два метода нашего класса, относящиеся к мыши. Метод `mouseDragged()` автоматически вызывается для слушателя, чтобы получать события, генерируемые при перетаскивании, то есть при перемещении мыши с любой нажатой кнопкой. Метод `mouseMoved()` вызывается каждый раз, когда пользователь перемещает мышь внутри заданной области без нажатия кнопки. В нашем примере мы разместили эти методы в классе `HelloComponent2` и заставили его зарегистрировать себя как слушателя. Это вполне подходит для нашего простого компонента перетаскивания текста. Но в хорошем стиле программирования слушатели событий реализуются в виде *классов-адаптеров*, обеспечивающих наглядное отделение графического интерфейса от «бизнес-логики». Эту тему мы рассмотрим в главе 10.

Наш метод `mouseMoved()` тривиален: он не делает ничего. Мы игнорируем простые перемещения мыши и уделяем все внимание перетаскиванию. Метод `mouseDragged()` более содержателен. Он многократно вызывается оконной системой для передачи обновляющейся информации о положении указателя мыши. Вот как он выглядит:

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

В первом аргументе метода `mouseDragged()` ему передается объект `MouseEvent` с именем `e`; он содержит всю необходимую информацию о событии. Мы запрашиваем у объекта `MouseEvent` координаты `x` и `y` для текущей позиции мыши, вызывая его методы `getX()` и `getY()`. Координаты сохраняются в переменных экземпляра `messageX` и `messageY` для последующего использования.

Элегантность модели событий заключается в том, что вы обрабатываете только те виды событий, которые вас интересуют. Если вам не нужны события клавиатуры, вы не регистрируете для них слушатель. В этом случае пользователь может вводить что угодно, но вас это нисколько не беспокоит. Если слушатели для конкретного вида событий отсутствуют, Java даже не генерирует это событие. В результате обработка событий выполняется очень эффективно¹.

Пока мы обсуждаем события, следует упомянуть другое небольшое дополнение в `HelloJava2`:

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

Так мы сообщаем объекту `JFrame`, что при щелчке на кнопке закрытия окна надо завершить приложение. Это называется операцией по умолчанию, потому что эта операция, как и почти все остальные взаимодействия с графическим интерфейсом, управляется событиями. Мы могли бы зарегистрировать слушатель для окна, чтобы получать уведомления о щелчках на кнопке закрытия и затем выполнять любые действия по своему усмотрению, но этот удобный метод обрабатывает самые типичные случаи.

Мы обошли вниманием пару вопросов: как система узнает, что наш класс содержит необходимые методы `mouseDragged()` и `mouseMoved()`? (Откуда взялись эти имена?) И зачем нужен метод `mouseMoved()`, который ничего не делает? Ответы на эти вопросы связаны с интерфейсами. Тему интерфейсов мы скоро рассмотрим, а пока разберемся с тем, какова роль метода `repaint()`.

¹ С обработкой событий в Java 1.0 дело обстояло совершенно иначе. Тогда в Java еще не было концепции слушателей, а вся обработка событий осуществлялась переопределением методов в базовых классах GUI. Это было неэффективно, а код усложнялся из-за размножения компонентов крайне узкого назначения.

Метод `repaint()`

После изменения координат надписи (в результате перетаскивания) компонент `HelloComponent2` должен перерисовать себя. Для этого мы вызываем метод `repaint()`, который сообщает системе, что через некоторое время она должна будет перерисовать изображение в окне. Мы не можем напрямую вызвать метод `paintComponent()`, даже если бы захотели, потому что у нас нет графического контекста для передачи этому методу.

Метод `repaint()` класса `JComponent` обеспечивает перерисовку компонента. Он заставляет оконную систему Java запланировать вызов нашего метода `paintComponent()` в ближайший возможный момент; при этом Java предоставляет необходимый объект `Graphics` (рис. 2.16).

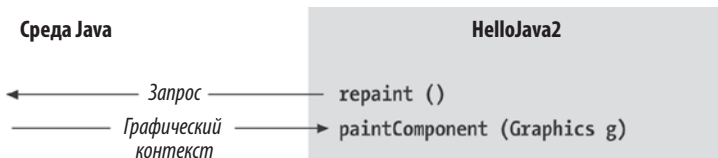


Рис. 2.16. Вызов метода `repaint()`

Не надо считать такой режим выполнения операции неудобством, вызванным отсутствием подходящего графического контекста. Главное преимущество заключается в том, что процессом перерисовки занимается кто-то другой, пока мы можем свободно делать что-то другое. Система Java содержит отдельный специализированный поток, который обрабатывает все запросы `repaint()`. Она может планировать и консолидировать запросы `repaint()` по мере необходимости, что позволяет избежать перегрузки оконной системы в ситуациях с интенсивной перерисовкой (например, при прокрутке). Есть и другое преимущество: вся функциональность перерисовки инкапсулируется в методе `paintComponent()`; по крайней мере, у вас не будет соблазна распределить ее по всему приложению.

Интерфейсы

А теперь пора ответить на вопрос, который был обойден ранее: как система узнает, что при возникновении события мыши надо вызвать метод `mouseDragged()`? Может быть, `mouseDragged()` — какое-то специальное имя, которое должно присваиваться методу обработки события? Нет, это не так. Ответ на вопрос требует рассмотрения интерфейсов — одной из самых важных концепций языка Java.

Первый признак использования интерфейса встречается в строке кода, представляющей класс `HelloComponent2`. Там мы указываем, что этот класс реализует интерфейс `MouseMotionListener`:

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

По сути интерфейс — это список методов, которые должен содержать класс. Этот конкретный интерфейс требует, чтобы в классе были методы с именами `mouseDragged()` и `mouseMoved()`. Интерфейс ничего не говорит о том, что должны делать эти методы (например, `mouseMoved()` не делает ничего). Он требует, чтобы методы получали `MouseEvent` в аргументе и не возвращали никаких значений (напомним, отсутствие значения — это `void`).

Иными словами, интерфейс — это своего рода «контракт» между программистом и компилятором. Указывая, что ваш класс реализует интерфейс `MouseMotionListener`, вы тем самым говорите, что эти методы будут доступны для вызова из других частей системы. Если вы не предоставите эти методы, произойдет ошибка компиляции.

Влияние интерфейса на программу этим не ограничивается. Интерфейс также действует как класс. Например, метод может вернуть объект `MouseMotionListener` или получить `MouseMotionListener` в аргументе. Когда вы ссылаетесь на объект по имени интерфейса, это означает, что реальный класс объекта вас не интересует; требуется только одно — чтобы класс реализовал этот интерфейс. Примером служит метод `addMouseMotionListener()`: его аргументом должен быть объект, реализующий интерфейс `MouseMotionListener`. Передаваемым аргументом является `this`, то есть сам объект `HelloComponent2`. Тот факт, что он является экземпляром класса `JComponent`, роли не играет; это может быть `Cookie`, `Aardvark` или любой другой класс, который вы придумаете. Важно лишь то, что он реализует `MouseMotionListener`, а следовательно, объявляет, что он содержит два метода с заданными именами. Вот почему метод `mouseMoved()` необходим: он ничего не делает, но интерфейс `MouseMotionListener` требует, чтобы этот метод присутствовал.

В Java есть много интерфейсов, определяющих, что должны делать классы. Идея «контракта» между компилятором и классом очень важна. Есть много ситуаций вроде описанной выше, когда вас не интересует, к какому классу относится объект; вам важно лишь то, что он обладает некоторой функциональностью, например прослушиванием событий мыши. Интерфейсы позволяют работать с объектами на уровне их функциональности, не зная их фактических типов. Концепция интерфейсов чрезвычайно важна для Java как объектно-ориентированного языка. Мы подробно рассмотрим интерфейсы в главе 5.

Из главы 5 вы также узнаете, что интерфейсы предоставляют лазейку для обхода правила Java, согласно которому любой новый класс может расширять только один родительский класс («одиночное наследование»). Да, класс в Java может расширять только один класс, но зато может реализовать сколько угодно интерфейсов. Интерфейсы могут использоваться как типы данных, могут реализовывать другие интерфейсы (но не классы) и могут наследоваться классами (если класс А реализует интерфейс В, то subclasses А также реализуют В). Принципиальное отличие заключается в том, что классы не наследуют методы из интерфейсов; интерфейс всего лишь определяет, какие методы должен содержать класс.

До свидания... и снова здравствуйте!

Пришло время попрощаться с приложением `HelloJava`. Надеемся, вы получили представление о некоторых возможностях языка и об основных правилах написания и запуска программ. Это краткое введение помогло вам разобраться в том, как программировать на Java. Даже если вы что-то не поняли — крепитесь. Все основные темы, упомянутые в этой главе, будут подробно рассмотрены в последующих главах. А пока вы прошли «боевое крещение» и познакомились с важнейшими концепциями и терминами, которые сразу узнаете при следующей встрече.

Мы еще вернемся к `HelloJava`, а прямо сейчас расскажем о наборе инструментов, с которыми работают программисты. В главе 3 подробно описаны инструменты, которые вы уже немного знаете (например, `javac`), а также другие важные утилиты. Приготовьтесь, вас ждет встреча с лучшими друзьями Java-разработчика!

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javalib>

ГЛАВА 3

Рабочие инструменты

Почти наверняка вы будете разрабатывать Java-приложения в среде IDE: это может быть Eclipse, VS Code или предпочитаемая авторами книги IntelliJ IDEA. Тем не менее все основные инструменты, необходимые для создания приложений Java, включены в пакет JDK, который вы, скорее всего, уже загрузили с сайта Oracle (см. раздел «Установка JDK», с. 54)¹. В этой главе мы рассмотрим базовые средства командной строки, которые нужны для компиляции, запуска и упаковки приложений Java. В составе JDK есть и многие другие средства разработчика, о которых мы расскажем впоследствии.

За дополнительной информацией об IntelliJ IDEA и за инструкциями по загрузке всех примеров книги в виде проекта обращайтесь к приложению, с. 497.

Среда JDK

После установки Java главная команда `java` может появиться в пути запуска автоматически. Но многие другие команды, входящие в JDK, могут быть недоступными, если вы не включите каталог Java `bin` в свой путь запуска. Ниже показано, как сделать это в Linux, macOS и Windows. Конечно, эти примеры надо скорректировать, указав точный путь к Java на вашем компьютере.

Для Linux:

```
export JAVA_HOME=/usr/lib/jvm/java-12-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin
```

Для macOS:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-12.jdk/Contents/Home
export PATH=$PATH:$JAVA_HOME/bin
```

¹ Кроме Oracle, есть и другие поставщики JDK; при желании вы можете найти их в интернете и сравнить разные дистрибутивы.

Для Windows:

```
set JAVA_HOME=c:\Program Files\Java\jdk12
set PATH=%PATH%;%JAVA_HOME%\bin
```

В macOS ситуация может быть более запутанной, потому что в последних версиях macOS есть встроенные «заглушки» для команд Java. При попытке выполнить любую из этих команд операционная система предложит вам загрузить Java. Вы можете заранее загрузить OpenJDK с сайта Oracle; инструкции приведены в разделе «Инструменты и среда Java», с. 53.

Если вы сомневаетесь в том, какая версия Java установлена на компьютере, выполните команды `java` и `javac` с флагом `-version`:

```
java -version
# openjdk version "12" 2019-03-19
# OpenJDK Runtime Environment (build 12+33)
# OpenJDK 64-Bit Server VM (build 12+33, mixed mode, sharing)

javac -version
# javac 12
```

Виртуальная машина Java

Виртуальная машина (VM) Java — это программа, которая реализует исполнительную систему (интерпретатор) Java и выполняет Java-приложения. Это может быть автономная программа (например, команда `java` из JDK) или код, встроенный в другое приложение, такое как браузер. Чаще всего интерпретатор представляет собой платформенное приложение, версии которого поставляются для каждой конкретной платформы; он запускает другие приложения, написанные на Java. Многие компиляторы Java и IDE написаны именно на Java для максимальной портируемости и расширяемости. Например, IDE Eclipse является чистым Java-приложением.

Виртуальная машина (VM) отвечает за все, что происходит при работе Java-приложений. Она загружает файлы классов Java, проверяет классы из ненадежных источников и выполняет скомпилированный байт-код. Она управляет памятью и системными ресурсами. Кроме того, лучшие реализации VM выполняют динамическую оптимизацию, компилируя байт-код Java в платформенный машинный код компьютера.

Запуск приложений Java

В каждом отдельном приложении Java должен быть стартовый класс, содержащий метод с именем `main()`; код этого метода выполняется сразу после запуска.

Запустите виртуальную машину и передайте ей имя этого класса в командной строке, чтобы запустить приложение. Также в командной строке можно указать системные параметры для интерпретатора и передаваемые приложению аргументы:

```
$ java [параметры для интерпретатора] имя_класса [аргументы]
```

Класс должен быть указан с полным именем, включающим имя пакета (если оно есть). Расширение файла `.class` в имя не включается. Несколько примеров:

```
$ java animals.birds.BigBird
$ java MyTest
```

Интерпретатор ищет стартовый класс в `classpath`, то есть в списке каталогов и архивных файлов, в которых хранятся классы. Список `classpath` будет подробно рассмотрен в следующем разделе. Его можно задать либо в *переменной среды*, либо в параметре командной строки `-classpath`. Если заданы оба варианта, то используется параметр командной строки.

Также команда `java` может запускать приложения из исполняемых архивов Java, имеющих формат JAR:

```
$ java -jar spaceblaster.jar
```

В таких случаях JAR-файл должен содержать метаданные, в которых указано имя стартового класса с методом `main()`, а в качестве `classpath` служит сам JAR-файл.

После загрузки стартового класса и выполнения его метода `main()` приложение может ссылаться на другие классы, запускать дополнительные потоки и создавать пользовательский интерфейс и другие структуры, как показано на рис. 3.1.

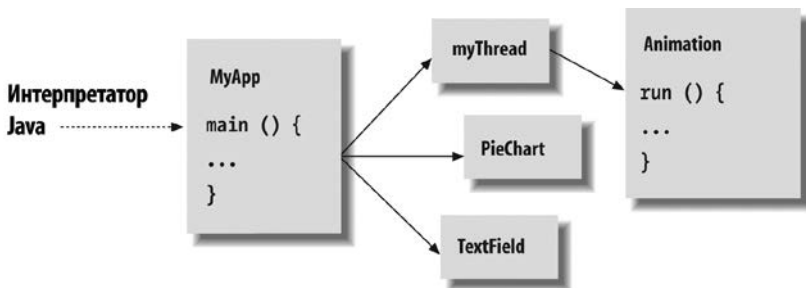


Рис. 3.1. Запуск приложения Java

Метод `main()` должен иметь правильную *сигнатуру метода*. Сигнатура метода — это набор данных, определяющих метод. Она включает имя метода, аргументы и возвращаемый тип, а также модификаторы типа и видимости.

Метод `main()` должен быть открытым (`public`) и статическим (`static`); он должен получать в аргументе массив объектов `String` и не возвращать никакого значения (`void`):

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

Тот факт, что метод `main()` является открытым и статическим, означает лишь то, что он доступен глобально и может напрямую вызываться по имени. Смысл модификаторов видимости (таких, как `public`) и модификатора `static` мы объясним в главах 4 и 5.

Единственный аргумент метода `main()` — массив объектов `String`. Он содержит аргументы командной строки, переданные приложению. Имя этого параметра ни на что не влияет; важен только тип. В Java содержимым `myArgs` является массив (подробнее о массивах — в главе 4). В Java массивам известно, сколько элементов они содержат, и они могут предоставить эту информацию:

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

`myArgs[0]` — первый аргумент командной строки (и т. д.).

Интерпретатор Java продолжает работать, пока метод `main()` исходного файла класса не вернет управление и пока не будут завершены все запущенные им потоки. (Подробнее о потоках — в главе 9.) Специальные потоки, определенные как потоки-демоны, завершаются автоматически — при завершении всех остальных частей приложения.

Системные параметры

Хотя из кода Java можно читать системные переменные среды, они плохо подходят для настройки конфигурации приложения. Вместо этого Java позволяет передать приложению значения произвольных системных параметров при запуске виртуальной машины. Системные параметры — это простые пары строк «имя — значение», которые приложение может запросить, вызвав статический метод `System.getProperty()`. Используйте системные параметры как более структурированную и портируемую альтернативу аргументам командной строки и переменным среды, когда надо передать приложению при запуске основную информацию о конфигурации. Каждый системный параметр передается интерпретатору в командной строке с ключом `-D`, за которым следует пара `имя=значение`. Пример:

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

Затем к значению свойства `street` можно обратиться следующим образом:

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

Приложение может получить параметры своей конфигурации множеством других способов, в том числе прочитать из файлов или получить по сети во время выполнения.

Classpath

Путь (path) — это понятие, знакомое каждому, кто работал в DOS или Unix. Так называется переменная среды, предоставляющая приложениям список мест для поиска некоторого ресурса. Самый распространенный пример — путь к исполняемому файлу. В Unix переменная среды PATH содержит список каталогов, в которых последовательно производится поиск при вводе пользователем команды. Похожая переменная среды Java CLASSPATH содержит список каталогов для поиска пакетов и классов Java. Как интерпретатор, так и компилятор Java используют CLASSPATH.

В списке `classpath` могут быть указаны каталоги и JAR-файлы. Java также поддерживает архивы в традиционном формате ZIP, но JAR и ZIP в действительности имеют одинаковый формат. JAR-файлы — это обычные архивы, в которых есть служебные файлы (метаданные) с описанием содержимого архивов. JAR-файлы создаются утилитой `jar` из JDK. Есть много общедоступных программ для создания ZIP-архивов, с помощью которых тоже можно создавать и просматривать JAR-файлы. Формат архива позволяет распространять большие группы классов и их ресурсов в одном файле. Исполнительная система Java автоматически извлекает отдельные файлы классов из архива по мере необходимости.

Формат `classpath` зависит от конкретной операционной системы. В Unix-подобных системах (включая macOS) задается переменная среды CLASSPATH, которая содержит список каталогов и архивных файлов классов, разделенных двоеточиями:

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

В этом примере задается список `classpath`, состоящий из трех путей: к каталогу в домашнем каталоге пользователя, к JAR-файлу в каталоге другого пользователя и к текущему каталогу, который всегда обозначается точкой (`.`). Последний компонент `classpath` (текущий каталог) пригодится при экспериментах с классами.

В Windows переменная среды CLASSPATH содержит список каталогов и архивных файлов классов, разделенных символами «точка с запятой» (`;`):

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

Программа запуска Java и другие утилиты командной строки знают, где искать фундаментальные классы (классы, включаемые в каждую реализацию Java).

Например, классы пакетов `java.lang`, `java.io`, `java.net` и `javax.swing` являются фундаментальными и включать их в `classpath` не нужно.

Пути в списке `classpath` могут содержать групповые символы `*`, обозначающие все JAR-файлы в каталоге. Пример:

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

Чтобы найти другие классы, кроме фундаментальных, интерпретатор Java проводит поиск по элементам списка `classpath` в порядке их следования. При поиске каждый путь объединяется с полным именем класса. Для примера посмотрим, как выполняется поиск класса `animals.birds.BigBird`. Поиск в каталоге `/usr/lib/java` из `classpath` означает, что интерпретатор ищет файл `/usr/lib/java/animals/birds/BigBird.class`. Поиск в JAR-архиве `/home/vicky/myutils.jar` из `classpath` (или в аналогичном ZIP-архиве) означает, что интерпретатор ищет в этом архиве файл `animals/birds/BigBird.class`.

Для исполнительной системы `java` и компилятора `javac` список `classpath` можно задать параметром `-classpath` в командной строке:

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

Если не задать ни переменную среды `CLASSPATH`, ни параметр командной строки, то по умолчанию в качестве `classpath` используется текущий каталог (`.`); это означает, что исполнительной системе Java будут доступны файлы в текущем каталоге. Но если вы составите `classpath` и не включите в него текущий каталог, то эти файлы окажутся недоступными.

Подозреваем, что около 80% всех проблем, с которыми сталкиваются новички при изучении Java, связаны с `classpath`. Возможно, вам надо уделить особое внимание составлению и проверке `classpath` прямо сейчас. Если вы работаете в IDE, то эта среда может избавить вас (частично или полностью) от хлопот с `classpath`. Однако в перспективе вам надо будет хорошо понимать, какие пути есть в списке `classpath` и каково содержимое соответствующих каталогов и архивов во время работы ваших приложений. Выявлять проблемы с `classpath` вам поможет команда `javap`.

javap

Еще один полезный инструмент, о котором вам надо знать, — это команда `javap`. Она выводит на экран описание скомпилированного класса. Для этого не нужен исходный код класса и даже не нужно знать его точное местонахождение — достаточно, чтобы он входил в `classpath`. Например, следующая команда выводит информацию о классе `java.util.Stack`:

```
$ javap java.util.Stack
```

Результат вывода будет таким:

```
Compiled from "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);
}
```

Такая информация окажется очень полезной, если у вас нет другой документации; также она пригодится при отладке проблем с `classpath`. При помощи `javap` всегда можно определить, входит ли класс в `classpath`, и даже проверить его версию (многие проблемы возникают из-за дублирования классов в `classpath`). По-настоящему любознательные читатели могут запустить `javap` с параметром `-c`, чтобы команда также вывела байт-код виртуальной машины для каждого метода в классе.

Модули

В Java 9 вместо традиционного списка `classpath` (который по-прежнему поддерживается) можно воспользоваться новым решением, основанным на модулях. Модули позволяют выполнять более детализированное и эффективное развертывание приложений, даже очень больших. Этот вариант требует дополнительной настройки, поэтому в книге он не рассматривается, но важно знать, что почти все большие коммерческие приложения имеют модульную структуру. Ценную информацию и помощь в модульном структурировании больших проектов вы найдете в книге Пола Баккера (Paul Bakker) и Сандера Мака (Sander Mak) «Java 9 Modularity» (<https://www.oreilly.com/library/view/java-9-modularity/9781491954157>) — она будет полезной, если вы хотите распространять свои работы, не ограничиваясь простой отправкой исходного кода в общедоступные репозитории.

Компилятор Java

В этом разделе мы скажем несколько слов о программе `javac`, которая является компилятором Java из JDK. Эта программа полностью написана на Java, поэтому доступна на любой платформе с поддержкой исполнительной системы Java. Компилятор преобразует исходный код Java в скомпилированные классы, состоящие из байт-кода Java. Исходные файлы должны иметь расширение `.java`, а скомпилированные файлы классов — расширение `.class`. Каждый файл с ис-

ходным кодом считается отдельной единицей компиляции. Как будет показано в главе 5, все классы заданной единицы компиляции обладают рядом сходных признаков, например, относящихся к операторам `package` и `import`.

Компилятор `javac` разрешает использовать один открытый (`public`) класс в каждом файле и требует, чтобы имя этого файла совпадало с именем класса. Если имена файла и класса не совпадают, то `javac` выдает ошибку компиляции. Один файл может содержать несколько классов при условии, что только один из них объявлен открытым, а его имя совпадает с именем файла. Старайтесь не упаковывать слишком много классов в один исходный файл. Упаковка классов в файле `.java` только создает иллюзию связи между ними. В главе 5 рассказано о внутренних классах (`inner classes`) — таких, которые содержатся в других классах и интерфейсах.

Например, включите следующий фрагмент кода в файл с именем `BigBird.java`:

```
package animals.birds;

public class BigBird extends Bird {
    ...
}
```

Скомпилируйте его следующей командой:

```
$ javac BigBird.java
```

В отличие от интерпретатора Java, который получает в аргументе имя класса, `javac` должен получить в аргумента имя файла (с расширением `.java`), который надо скомпилировать. Эта команда создает файл `BigBird.class` в одном каталоге с исходным файлом. Хотя в данном примере удобно создать файл класса в одном каталоге с исходным файлом, в реальных приложениях файл класса обычно должен храниться в более подходящем месте из `classpath`.

Вы можете запускать `javac` с параметром `-d`, чтобы указывать определенный каталог для хранения файлов тех классов, которые генерирует `javac`. Указанный каталог используется в качестве корневого в иерархии классов, поэтому файлы `.class` помещаются в этот каталог или в один из его подкаталогов в зависимости от того, содержится ли класс в пакете. При необходимости компилятор создает промежуточные каталоги автоматически. Например, следующая команда создает файл `/home/vicky/Java/classes/animals/birds/BigBird.class`:

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

В одной команде `javac` можно указать несколько файлов `.java`; в этом случае компилятор создает файл класса для каждого исходного файла. Однако вам не надо перечислять другие классы, на которые ссылается ваш класс, если они находятся в `classpath` в форме исходного кода или в скомпилированном виде.

Во время компиляции класса Java обрабатывает все ссылки на другие классы, основываясь на `classpath`.

Компилятор Java «умнее» типичных компиляторов: он заменяет часть функциональности утилиты `make`. Например, `javac` всегда сравнивает время изменения исходного файла и существующего файла класса (если таковой есть), а затем при необходимости перекомпилирует исходный файл. Скомпилированный класс Java хранит информацию об исходном файле, из которого он был скомпилирован, и пока исходный файл остается доступным, компилятор может перекомпилировать его при необходимости. Если класс `BigBird` из предыдущего примера ссылается на другой класс (например, `animals.furry.Grover`), то `javac` ищет исходный файл `Grover.java` в пакете `animals.furry` и при необходимости его перекомпилирует, чтобы класс `Grover.class` оставался актуальным.

Однако по умолчанию `javac` проверяет только те исходные файлы, на которые напрямую ссылаются другие исходные файлы. Таким образом, если у вас есть устаревший файл класса, на который ссылается только актуальный файл класса (но не исходный код), он не будет обнаружен и перекомпилирован. Из-за этого (и по многим другим причинам) в большинстве крупных проектов для управления сборкой, упаковкой и другими подобными задачами используются полноценные утилиты сборки, например Gradle.

Наконец, важно заметить, что `javac` может скомпилировать приложение даже в том случае, если некоторые его классы доступны только в скомпилированных (двоичных) версиях. Наличие исходного кода всех объектов не обязательно. Файлы классов Java, как и исходные файлы, содержат всю информацию о типах данных и сигнатуры всех методов, поэтому компиляция с двоичными файлами классов настолько же безопасна по отношению к типам и исключениям, как и компиляция с исходным кодом.

Первые эксперименты с Java

В Java 9 появилась утилита `jshell`, которая позволяет опробовать фрагменты кода Java и немедленно увидеть результаты. `jshell` является оболочкой REPL (Read-Evaluate-Print Loop, то есть «цикл чтение — вычисление — вывод»). Такие оболочки есть во многих языках, и до выхода Java 9 появилось много сторонних разработок, но встроенной программы в JDK тогда не было. Мы уже упоминали о `jshell`; теперь познакомимся поближе с возможностями этой программы.

Откройте окно терминала или командной строки вашей операционной системы либо откройте вкладку терминала в IntelliJ IDEA, как показано на рис. 3.2. Введите команду `jshell` в командной строке. Вы получите информацию о версии и краткое напоминание о том, как пользоваться справкой из REPL.

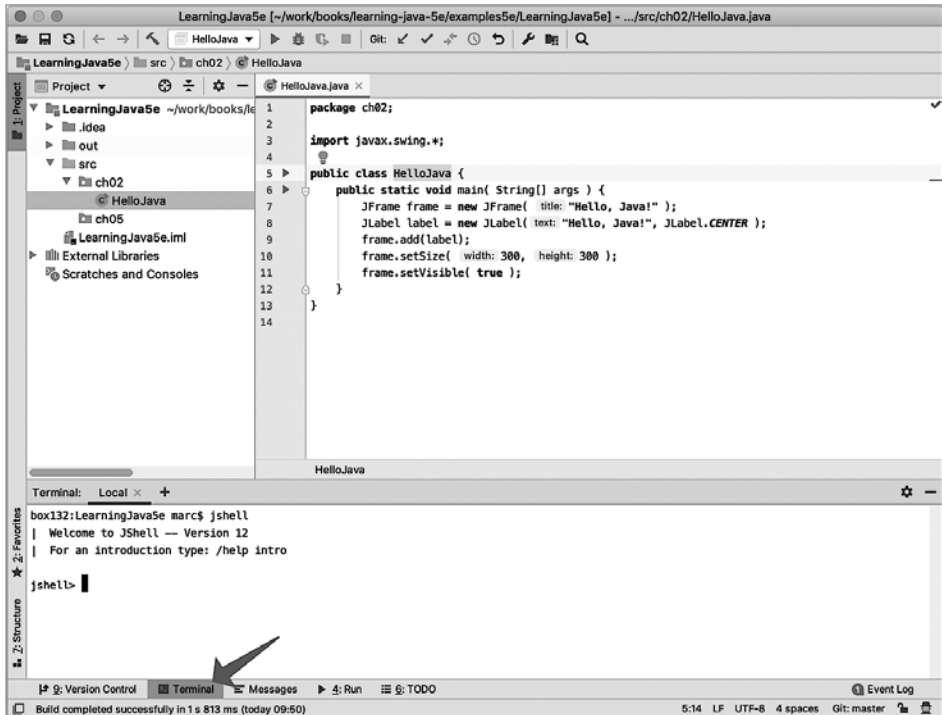


Рис. 3.2. Запуск jshell из IDEA

Попробуйте выполнить команду `/help intro` для вывода справки:

```
| Welcome to JShell -- Version 12
| For an introduction type: /help intro
```

```
jshell> /help intro
```

```
|
|                                     intro
|                                     =====
|
| The jshell tool allows you to execute Java code, getting immediate results.
| You can enter a Java definition (variable, method, class, etc),
| like: int x = 8
| or a Java expression, like: x + x
| or a Java statement or import.
| These little chunks of Java code are called 'snippets'.
|
| There are also the jshell tool commands that allow you to understand and
| control what you are doing, like: /list
|
| For a list of commands: /help
```

`jshell` — это мощный инструмент, и для целей этой книги нам не понадобятся все его возможности. Но в последующих главах он будет нужен, чтобы пробовать фрагменты кода Java и быстро вносить изменения по ходу дела. Вспомните пример `HelloJava2` из раздела «HelloJava2: продолжение», с. 79. Вы можете создать элементы пользовательского интерфейса (такие, как `JFrame`) прямо в REPL, а потом работать с ними — и немедленно получать обратную связь! Вам не придется сохранять, компилировать, запускать, редактировать, сохранять, компилировать, запускать... Давайте попробуем:

```
jshell> JFrame frame = new JFrame( "HelloJava2" )
| Error:
| cannot find symbol
|   symbol:   class JFrame
| JFrame frame = new JFrame( "HelloJava2" );
| ^----^
| Error:
| cannot find symbol
|   symbol:   class JFrame
| JFrame frame = new JFrame( "HelloJava2" );
|                               ^----^
```

Не получилось! Оболочка `jshell` хорошо сделана и обладает широкой функциональностью, но она все понимает буквально. Помните: если вы хотите использовать класс, не включенный в пакет по умолчанию, то вам надо его импортировать. Это относится не только к исходным файлам Java, но и к работе с `jshell`. Попробуем еще раз:

```
jshell> import javax.swing.*

jshell> JFrame frame = new JFrame( "HelloJava2" )
frame ==> javax.swing.JFrame[frame0,0,23,0x0,invalid,hidden ... led=true]
```

Уже лучше. Возможно, немного странно, но явно лучше. Наш объект `JFrame` был успешно создан. После стрелки `==>` сообщается дополнительная информация о `JFrame`: размер (0x0) и позиция на экране (0,23). Для других типов объектов выводится другая информация. Теперь укажем ширину и высоту окна, как это делалось ранее, и отобразим окно на экране:

```
jshell> frame.setSize(300,200)

jshell> frame.setLocation(400,400)

jshell> frame.setVisible(true)
```

Вы увидите, что на экране появилось окно! Оно выглядит вполне современно — примерно так, как показано на рис. 3.3.



Рис. 3.3. Создание объекта JFrame из jshell

Не беспокойтесь о том, что вы можете допустить ошибку в REPL. Появится сообщение об ошибке, но вы сможете исправить то, что сделали неправильно, а потом продолжить работу. Представьте, например, что вы допустили опечатку при попытке изменить размер окна:

```
jshell> frame.setsize(300,300)
| Error:
| cannot find symbol
|   symbol:   method setsize(int,int)
| frame.setsize(300,300)
| ^-----^
```

В языке Java учитывается регистр символов, поэтому `setSize()` — это не то же самое, что `setsize()`. `jshell` выдает ту же информацию об ошибке, что и компилятор Java, но представляет ее в другом виде. Исправьте ошибку, и вы увидите, что окно немного увеличилось (рис. 3.4).



Рис. 3.4. Изменение размера окна

Потрясающе! По правде говоря, это было скорее эффектно, чем полезно, но все только начинается. Добавим текст при помощи класса `JLabel`:

```
jshell> JLabel label = new JLabel("Hi jshell!")
label ==>
javax.swing.JLabel[,
0,0,0x0, ...
rticalTextPosition=CENTER]
```

```
jshell> frame.add(label)
$8 ==>
javax.swing.JLabel[,0,0,0x0, ...
text=Hi, ...]
```

Неплохо, но почему надпись не появилась в окне? Этот вопрос будет подробнее рассмотрен в главе, посвященной пользовательским интерфейсам, но суть в том, что Java может накапливать графические изменения, прежде чем отображать их на экране. Этот прием бывает невероятно эффективным, но иногда он может заставить вас врасплох. Заставим окно перерисовать себя (рис. 3.5):

```
jshell> frame.revalidate()

jshell> frame.repaint()
```



Рис. 3.5. Добавление объекта `JLabel` в окно

Теперь надпись видна. Некоторые действия автоматически инициируют вызов `revalidate()` или `repaint()`. Например, любой компонент, уже добавленный в окно до того, как оно появилось на экране, немедленно становится видимым

с появлением окна. Надпись можно удалить тем же способом, которым она была добавлена. Снова посмотрите, что произойдет при изменении размера окна сразу же после удаления надписи (рис. 3.6):

```
jshell> frame.remove(label) // Как и в случае с add(), изображение
                          // не изменяется немедленно
jshell> frame.setSize(400,150)
```



Рис. 3.6. Удаление надписи и изменение размеров окна

Видите? Окно стало более узким, а надпись исчезла — и все это без принудительной перерисовки. Мы еще поработаем с GUI-элементами в дальнейших главах, а пока попробуем провести еще один эксперимент, который показывает, как легко опробовать новые идеи и методы, которые вы обнаружили в документации. Например, текст надписи можно выровнять по центру; примерный результат показан на рис. 3.7:

```
jshell> frame.add(label)
$45 ==>
javax.swing.JLabel[,0,0,300x278,...,
text=Hi jshell!,...]

jshell> frame.revalidate()

jshell> frame.repaint()

jshell> label.setHorizontalAlignment(JLabel.CENTER)
```



Рис. 3.7. Выравнивание текста по центру надписи

Вероятно, для вас это был очередной головокружительный полет с несколькими фрагментами кода, которые пока могут выглядеть загадочно. Например, почему слово `CENTER` написано в верхнем регистре? И почему перед выравниванием по центру указывается имя класса `JLabel`? Надеемся, что после ввода всех этих строк (даже если вы допустили пару мелких ошибок и исправили их) вы захотели узнать больше. Мы стараемся, чтобы у вас было все необходимое для собственных экспериментов, когда вы будете читать другие главы этой книги. Как и во многих других областях, в программировании намного полезнее сделать что-то своим руками, чем прочитать об этом!

JAR-файлы

Архивные JAR-файлы — это своего рода «чемоданы» из мира Java. Они предоставляют стандартные и портируемые средства для упаковки всех частей приложения Java в компактный пакет, который легко распространять и устанавливать. В JAR-файл можно поместить что угодно: файлы классов Java, упорядоченные объекты, файлы данных, графику, звук и т. д. JAR-файл может содержать одну или несколько цифровых подписей, удостоверяющих его целостность и подлинность. Подписью может снабжаться как файл в целом, так и его отдельные элементы.

Исполнительная система Java может загружать файлы напрямую из архива, указанного в `CLASSPATH`, как было описано ранее. Файлы из JAR-файла, не являющиеся файлами классов (данные, изображения и т. д.), загружаются из `classpath` в ваше приложение при помощи метода `getResource()`. При этом вашему коду не надо знать, является ли ресурс простым файлом или элементом JAR-архива. Независимо от того, является ли файл класса или файл данных элементом JAR-файла или отдельным файлом в `classpath`, вы всегда можете ссылаться на него стандартным способом и предоставить загрузчику классов Java самостоятельно определить его местонахождение.

Сжатие

Данные, хранящиеся в JAR-файлах, сжимаются стандартным алгоритмом ZIP. Сжатие значительно ускоряет передачу классов по сети. Беглый просмотр стандартного дистрибутива Java показывает, что типичный класс сжимается приблизительно на 40%. Текстовые файлы, содержащие английские слова (например, файлы HTML или ASCII), часто сжимаются до одной десятой части своего исходного размера и даже менее. (Но графические файлы при сжатии обычно не уменьшаются, потому что самые распространенные графические форматы сами реализуют сжатие.)

При самостоятельной работе с Java вы также можете встретить устаревший формат архивов Pack200, оптимизированный для байт-кода. Во многих случаях он сжимает классы в четыре раза сильнее, чем ZIP. Мы поговорим о нем далее в этой главе.

Утилита `jar`

Утилита `jar`, включенная в JDK, — это простой инструмент для создания и чтения JAR-файлов. Ее пользовательский интерфейс не назовешь удобным. Он воспроизводит интерфейс команды Unix `tar` (Tape ARchive). Если вы работали с `tar`, то следующие «заклинания» покажутся вам знакомыми:

- `jar -cvf jarFile путь [путь] [...]`
Эта команда создает архив с именем `jarFile`, содержащий файлы по заданному пути (или по нескольким путям).
- `jar -tvf jarFile [путь] [...]`
Эта команда показывает содержимое архива `jarFile`, опционально — только по заданному пути (или по нескольким путям).
- `jar -xvf jarFile [путь] [...]`
Эта команда распаковывает содержимое архива `jarFile`, опционально — только по заданному пути (или по нескольким путям).

В этих командах флаги `c`, `t` и `x` сообщают `jar`, какую операцию надо выполнить: создание архива, просмотр содержимого архива или извлечение файлов из архива. Флаг `f` означает, что следующий аргумент содержит имя JAR-файла для выполнения операции. Необязательный флаг `v` приказывает `jar` выводить расширенную информацию о файлах; в этом режиме выводится информация о размерах файлов, времени изменения и степени сжатия.

Последующие элементы командной строки (то есть все, кроме символов, которые указывают `jar`, что и с какими файлами делать) интерпретируются как имена элементов архива. Если вы создаете архив, то перечисленные файлы и каталоги включаются в этот архив. При извлечении из архива извлекаются только перечисленные файлы. Если файлы не указаны, то `jar` распаковывает все содержимое архива.

Допустим, вы только что доделали свою новую игру `Spaceblaster`. Все файлы, связанные с игрой, хранятся в трех каталогах. Классы Java хранятся в каталоге `spaceblaster/game`, каталог `spaceblaster/images` содержит игровую графику, а каталог `spaceblaster/docs` — игровые данные. Все эти составляющие можно упаковать в архив следующей командой:

```
$ jar -cvf spaceblaster.jar spaceblaster
```

Так как был запрошен расширенный вывод, утилита `jar` сообщает, что она делает в данный момент:

```
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%)
```

Это значит, что `jar` создает архив `spaceblaster.jar` с содержимым каталога `spaceblaster`, добавляя в этот архив все каталоги и файлы из `spaceblaster`. В расширенном режиме `jar` выводит информацию об экономии места, достигнутой при сжатии файлов в архиве.

Архив распаковывается следующей командой:

```
$ jar -xvf spaceblaster.jar
```

Аналогичным образом из архива извлекаются отдельные файлы или каталоги:

```
$ jar -xvf spaceblaster.jar filename
```

Впрочем, обычно незачем распаковывать JAR-файл для использования его содержимого; Java умеет извлекать файлы из таких архивов автоматически. Содержимое JAR-файла выводится следующей командой:

```
$ jar -tvf spaceblaster.jar
```

Ниже приведен результат. Выводится список всех файлов с указанием их размеров и времени 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

Команда `jar` автоматически включает в архив каталог с именем `META-INF`. Каталог `META-INF` содержит файлы, описывающие содержимое JAR-файла. Он всегда содержит как минимум один файл: `MANIFEST.MF`. Файл `MANIFEST.MF` может содержать «упаковочный список» с именами файлов в архиве и набором атрибутов, определяемых пользователем, для каждой записи.

Манифест — это текстовый файл с набором строк в формате «ключевое слово: значение». По умолчанию манифест почти пуст и содержит только информацию о версии JAR-файла:

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

Также JAR-файлы могут снабжаться цифровыми подписями. В этом случае в манифест включается контрольная сумма (дайджест, хеш-код) каждого архивируемого элемента, а каталог `META-INF` содержит файлы цифровой подписи для элементов в архиве:

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

Вы можете добавить в манифест собственную информацию, определив собственный дополнительный файл манифеста при создании архива. Это одно из возможных мест для хранения других простых сведений о файлах в архиве (например, номера версии или информации об авторских правах).

Попробуйте создать файл со следующими строками в формате «ключевое слово: значение»:

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

Чтобы добавить эту информацию в манифест нашего архива, сохраните ее в файле с именем `myManifest.mf` и введите следующую команду `jar`:

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

Дополнительный флаг `m` указывает, что `jar` должен читать дополнительную информацию манифеста из файла, заданного в командной строке. Но как `jar` различит два файла, указанных в командной строке? Так как `m` предшествует `f`, утилита ожидает найти имя файла манифеста перед именем создаваемого JAR-файла. Вы думаете, что это сомнительное решение? Вы правы: если перечислить имена файлов в неправильном порядке, то `jar` выполнит неправильную операцию.

Приложение может запросить информацию манифеста из JAR-файла при помощи класса `java.util.jar.Manifest`.

Создание исполняемого JAR-файла

Кроме атрибутов, в файл манифеста можно включить несколько специальных значений. Одно из них, `Main-Class`, позволяет указать стартовый класс с методом `main()` для приложения, содержащегося в JAR:

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

Если вы добавите эту строку в манифест JAR-файла (при помощи флага `m`, описанного выше), то приложение можно будет запустить прямо из JAR-файла:

```
$ java -jar spaceblaster.jar
```

Некоторые GUI-интерфейсы поддерживают запуск приложения двойным щелчком на JAR-файле. Интерпретатор ищет в манифесте значение `Main-Class`, а затем загружает указанный класс как стартовый класс приложения. Похоже, эта функция часто меняется и поддерживается не во всех операционных системах, поэтому плохо подходит для тех случаев, когда вы собираетесь распространять свое приложение.

Утилита `pack200`

`Pack200` — это формат архивов, оптимизированный для хранения скомпилированных классов Java. `Pack200` — это эффективная «упаковка» для набора классов, исключая многие виды избыточной информации при хранении набора взаимосвязанных классов. По сути, этот способ сжатия заключается в том, что многие классы разбираются на части, которые потом снова эффективно собираются в один каталог. Затем применяется стандартный архиватор (такой, как `ZIP`), и в итоге получается выигрыш по сжатию в четыре раза и более. Исполнительная система Java не поддерживает формат `pack200`, поэтому на такие архивы нельзя ссылаться в `classpath`. Скорее это промежуточный формат, очень удобный для передачи по сети JAR-файлов, содержащих апплеты и другие веб-приложения.

`Pack200` был популярен для доставки апплетов в прежние времена, но апплеты ушли в небытие, а вместе с ними и сам формат потерял актуальность. Но вам, возможно, еще будут встречаться файлы `.pack.gz`, поэтому мы решили упомянуть инструменты, которыми вы будете пользоваться при работе с ними. Тем не менее сами эти инструменты были исключены в Java 14.

Для преобразования JAR-файлов в формат `pack200` и обратно служат команды `pack200` и `unpack200`, которые есть в `JDK` и `OpenJDK` до Java 13 включительно.

Например, для преобразования файла `foo.jar` в `foo.pack.gz` используйте команду `pack200`:

```
$ pack200 foo.pack.gz foo.jar
```

А так вы можете преобразовать `foo.pack.gz` в `foo.jar`:

```
$ unpack200 foo.pack.gz foo.jar
```

Обратите внимание: в процессе работы `pack200` полностью разбирает и реконструирует ваши классы — на уровне классов. Поэтому полученный файл `foo.jar` не будет совпадать с оригиналом с точностью до байта.

Следующий шаг

Как видите, экосистема Java содержит немало инструментов — они получили известность еще в то время, когда их впервые включили в JDK. Вам не придется сразу пользоваться всеми инструментами, упомянутыми в этой главе, так что не беспокойтесь, если их список показался вам слишком обширным. По мере дальнейшего изучения языка мы будем в основном пользоваться компилятором `javac`. Впрочем, даже тогда компилятор и другие инструменты будут удобно спрятаны за кнопками IDE. Мы просто рассказали о том, какие инструменты вам доступны, чтобы вы могли изучить их детальнее, если они вам понадобятся.

Надеемся, что теперь, когда вы уже видели часть арсенала, применяемого для создания и упаковки кода Java, вы хотите написать что-то существенное. В нескольких ближайших главах будут заложены основы для этого. Итак, за дело!

ГЛАВА 4

Язык Java

С этой главы начинается наш вводный курс синтаксиса языка Java. Поскольку все читатели отличаются по опыту программирования, нам было нелегко подобрать подходящий уровень для любой аудитории. Мы постарались найти баланс между подробным обзором для начинающих с примерами синтаксиса и предоставлением дополнительной информации для более опытных читателей, чтобы они могли быстро оценить различия между Java и другими языками. Так как синтаксис Java является производным от C, мы иногда приводим сравнения с функциональностью этого языка, но опыт программирования на C от вас не требуется. Далее, в главе 5 рассматривается объектно-ориентированная сторона Java, а также завершается обсуждение базовых возможностей языка. В главе 7 рассматриваются обобщения — механизм, который расширяет возможности типов и позволяет реализовать некоторые виды классов более гибко и безопасно. После этого мы займемся различными API языка Java и покажем, что можно делать с их помощью. Оставшаяся часть книги заполнена примерами решений полезных задач из нескольких областей. Если после вводных глав у вас останутся вопросы, то мы надеемся, что ответы на них вы найдете в примерах кода. И конечно, вы всегда сможете повышать квалификацию! Мы постараемся давать ссылки на другие ресурсы, которые помогут читателям, желающим продолжить изучение Java за рамками рассматриваемых тем.

Для тех, кто только начинает знакомство с программированием, постоянным попутчиком должен стать интернет. Бесчисленные сайты, статьи «Википедии», публикации в блогах и сообщество Stack Overflow помогут разобраться в конкретных темах и найти ответы на возникающие вопросы. Например, в этой книге рассматривается язык Java и объясняется, как начать писать на нем полезные программы, но книга не рассказывает о таких фундаментальных концепциях программирования, как *алгоритмы*. Впрочем, эти концепции естественным образом вошли в обсуждения и примеры кода, а ссылки на сторонние ресурсы позволят вам закрепить в памяти некоторые подробности и заполнить вынужденные пробелы.

Кодирование текста

Java — язык для интернета. Интернет-сообщество говорит и пишет на множестве разных человеческих языков, поэтому они должны поддерживаться и в программах, написанных на Java. Один из способов интернационализации в Java основан на наборе символов Unicode («Юникод») — международном стандарте, поддерживающем символы большинства существующих языков¹. В современных версиях Java символьные и строковые данные хранятся в соответствии со стандартом Unicode 6.0, в котором для представления каждого символа используются как минимум два байта.

Исходный код Java можно писать в формате Unicode и сохранять во множестве разных кодировок, от простой двоичной формы до символов Unicode, закодированных в ASCII. Это делает язык Java удобным для программистов из разных стран. Они могут использовать свои родные языки в именах классов, методов и переменных, а также в тексте, выводимом в приложениях.

Тип Java `char` и класс `String` обладают встроенной поддержкой Unicode. Во внутреннем представлении текст хранится в формате `char[]` или `byte[]`; но язык Java и API работают с ними прозрачно для вас, поэтому вам, как правило, не придется об этом думать. Unicode хорошо сочетается с ASCII (это самая распространенная кодировка символов для английского языка). Первые 256 символов определены как идентичные первым 256 символам кодировки ISO 8859-1 (Latin-1), так что Unicode фактически обладает обратной совместимостью с самыми распространенными англоязычными кодировками. Кроме того, кодировка UTF-8, одна из самых популярных для Unicode, сохраняет значения ASCII в однобайтовой форме. Она используется по умолчанию в скомпилированных файлах классов Java, так что английский текст в памяти хранится компактно.

Многие платформы не способны отображать все символы Unicode, определенные в настоящее время. Поэтому при написании программ на Java можно использовать специальные служебные последовательности (escape-последовательности) Unicode. Символ Unicode может представляться служебной последовательностью следующего вида:

```
\uxxxx
```

Здесь `xxxx` — это последовательность из 1–4 шестнадцатеричных цифр. Она обозначает символ Unicode в кодировке ASCII. Также эта форма используется в Java для вывода символов Unicode в тех средах, где они не поддерживаются.

¹ За дополнительной информацией о Unicode обращайтесь на сайт <http://www.unicode.org>. По иронии судьбы одним из алфавитов, обозначенных как «устаревшие и архаичные» и в настоящее время не поддерживаемых в Unicode, является яванский — исторический язык жителей острова Ява (Java).

Кроме того, в Java есть классы для чтения и записи символьных потоков Unicode в конкретных кодировках, включая UTF-8.

Unicode, как и многие долговечные стандарты в мире технологий, изначально проектировался с запасом: считалось, что ни в какой мыслимой кодировке не может быть более 64 К (65 536) символов. Но в итоге даже этого оказалось мало. Сейчас получили широкое распространение кодировки UTF-32. В частности, символы «эмодзи», часто встречающиеся в приложениях-мессенджерах, кодируются за пределами стандартного диапазона символов Unicode (например, канонический смайлик кодируется в Unicode значением 1F600). Java поддерживает для таких символов многобайтовые служебные последовательности UTF-16. Не все платформы, поддерживающие Java, совместимы с эмодзи; попробуйте запустить `jshell`, чтобы узнать, поддерживаются ли символы-эмодзи на вашем компьютере (рис. 4.1).

```
[jshell> System.out.println("\uD83D\uDE00")
😄

[jshell> System.out.println("\uD83D\uDCAF")
100

[jshell> System.out.println("\uD83C\uDF36")
🍦

jshell> █
```

Рис. 4.1. Вывод эмодзи в приложении «Терминал» из macOS

Впрочем, с такими символами необходима осторожность. Нам пришлось показать скриншот, чтобы вы увидели, что эти милые картинки отображаются в `jshell` на Mac. Но если запустить в этой же операционной системе десктопное приложение Java с классами `JFrame` и `JLabel`, о которых мы рассказывали в главе 3, то вы получите результат, показанный на рис. 4.2.

```
jshell> import javax.swing.*

jshell> JFrame f = new JFrame("Emoji Test")
f ==>
javax.swing.JFrame[frame0
,0,23,0x0,invalid,hidden ...
=true]

jshell> JLabel l = new JLabel("Hi \uD83D\uDE00")
l ==> javax.swing.JLabel[,
0,0,0x0,invalid,alignmentX=0. ...
=CENTER]
```



```
jshell> f.add(1)
$12 ==> javax.swing.JLabel[,0,0,0x0,invalid,alignmentX= ...
rticalTextPosition=CENTER]

jshell> f.setSize(300,200)

jshell> f.setVisible(true)
```

Это не означает, что вы не можете использовать или поддерживать эмодзи в своих приложениях, — просто надо знать о различиях в функциональности вывода. Следите за тем, чтобы не создать проблем у пользователей, которые будут запускать ваши приложения.

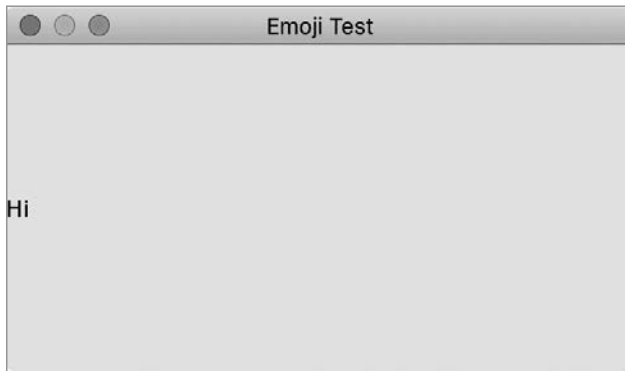


Рис. 4.2. Эмодзи не отображаются в JFrame

Комментарии

Java поддерживает как *блочные комментарии* в стиле C, заключенные в маркеры `/*` и `*/`, так и *строчные комментарии* в стиле C++, обозначаемые маркером `//`:

```
/* Это
    блочный комментарий
    (в несколько строк) */

// А это строчный комментарий (в одну строку)

// И это // тоже строчный комментарий
```

Блочные комментарии имеют как начальный, так и конечный маркер; они могут содержать большие объемы текста. Но блочные комментарии не должны быть вложенными, то есть попытка разместить блочный комментарий внутри другого блочного комментария будет ошибкой с точки зрения компилятора. Строчные

комментарии имеют только начальный маркер и завершаются в конце строки; дополнительные маркеры `//` внутри строки ни на что не влияют. Строчные комментарии удобны для включения кратких примечаний в методы; они не конфликтуют с блочными комментариями, так что вы можете закомментировать большие фрагменты кода, в которые они вложены.

Комментарии `javadoc`

Блочные комментарии, начинающиеся с символов `/**`, представляют собой специальные *doc-комментарии* (документирующие комментарии). Они предназначены для извлечения автоматическими генераторами документации (например, программой `javadoc` из JDK или системами контекстных подсказок во многих IDE). Doc-комментарий завершается конечным маркером `*/`, как и обычный блочный комментарий. Внутри doc-комментария все строки, начинающиеся с символа `@`, интерпретируются как специальные инструкции для генератора документации, передающие ему информацию об исходном коде. По общепринятым соглашениям в начале строк doc-комментария обычно добавляют символ `*`, как показано в следующем примере, но это не обязательно. Все начальные пробелы и символы `*` в строках игнорируются:

```
/**
 * Пожалуй, этот класс - самая потрясающая штука, которую вы
 * увидите в своей жизни. Позвольте мне пояснить свое
 * личное видение и причины для его создания.
 * <p>
 * Все началось еще тогда, когда я был ребенком и рос на улицах
 * Айдахо. Спрос на картофель был заоблачным, и жизнь была прекрасна...
 *
 * @see PotatoPeeler
 * @see PotatoMasher
 * @author John 'Spuds' Smith
 * @version 1.00, 19 Nov 2019
 */
class Potato {
```

Программа `javadoc` создает документацию классов в формате HTML, читая исходный код и извлекая встроенные комментарии и теги, начинающиеся с символа `@`. В данном примере теги включают в документацию класса информацию об авторе и версии. Теги `@see` генерируют гипертекстовые ссылки в документации класса.

Компилятор тоже проверяет doc-комментарии. В частности, его интересует тег `@deprecated`, который означает, что метод объявлен устаревшим и лучше не вызывать его в новых программах. Тот факт, что метод является устаревшим, отмечается в скомпилированном файле класса, поэтому при обнаружении в коде устаревших методов будут генерироваться предупреждения (даже если исходный код недоступен).

Дос-комментарии могут располагаться над определениями классов, методов и переменных, но некоторые теги могут быть неприменимы к некоторым из них. Например, тег `@exception` может применяться только к методам. В табл. 4.1 приведена сводка тегов, поддерживаемых в дос-комментариях.

Таблица 4.1. Теги в дос-комментариях

Тег	Описание	Применение
@see	Имя связанного класса	Класс, метод или переменная
@code	Содержимое исходного кода	Класс, метод или переменная
@link	Связанный URL-адрес	Класс, метод или переменная
@author	Имя автора	Класс
@version	Строка с версией	Класс
@param	Имя и описание параметра	Метод
@return	Описание возвращаемого значения	Метод
@exception	Имя и описание исключения	Метод
@deprecated	Объявление элемента как устаревшего	Класс, метод или переменная
@since	Версия API, в которой элемент был добавлен	Переменная

Теги javadoc как метаданные

Теги javadoc в дос-комментариях представляют собой *метаданные*, относящиеся к исходному коду; другими словами, они добавляют описательную информацию о структуре или содержании кода, которая формально не является частью приложения. Некоторые дополнительные инструменты расширяют концепцию тегов в стиле javadoc, позволяя включать в программы Java и другие виды метаданных, которые передаются вместе со скомпилированным кодом и могут легко использоваться приложением, воздействуя на его компиляцию или на логику работы во время выполнения. *Аннотации* Java предоставляют более формальный и расширяемый способ добавления метаданных в классы Java, методы и переменные. Эти метаданные также доступны на стадии выполнения.

Аннотации

Префикс `@` реализует в Java и другую функциональность, которая на первый взгляд напоминает функциональность тегов. *Аннотации* используются в Java для пометки контента, который должен обрабатываться специальным образом. Аннотации применяются к коду **за пределами** комментариев. Информация,

предоставляемая аннотацией, может быть полезна компилятору или IDE. Например, аннотация `@SuppressWarnings` заставляет компилятор (и часто также вашу IDE) скрывать предупреждения о таких потенциальных проблемах, как недоступный код. Когда мы займемся созданием более интересных классов в разделе «Нетривиальное проектирование классов», с. 189, вы увидите, что ваша IDE добавляет в код аннотации `@Overrides`. Они заставляют компилятор выполнять некоторые дополнительные проверки, которые способствуют написанию корректного кода и выявлению ошибок до того, как программа будет запущена вами или другими пользователями.

Вы можете создавать собственные аннотации для работы с другими инструментами и фреймворками. Хотя углубленное изучение аннотаций выходит за рамки этой книги, мы воспользуемся некоторыми очень удобными аннотациями при изучении веб-программирования в главе 12.

Переменные и константы

Итак, вы научились комментировать свой код. Это необходимо для того, чтобы он был наглядным и простым в сопровождении. Теперь нам пора сосредоточиться на самом коде, то есть на том, что компилируется. Все программирование сводится к работе с кодом. Почти во всех языках важная часть кода содержится в *переменных* и *константах*, которые упрощают работу программиста. В Java есть как переменные, так и константы. В переменных хранится та информация, которую вы собираетесь изменять и затем повторно использовать (или информация, неизвестная заранее: например, адрес электронной почты пользователя). В константах хранится информация, которая не должна изменяться. Примеры переменных и констант встречались вам даже в наших маленьких вводных программах. Вспомните простую графическую надпись из раздела «HelloJava» на с. 66:

```
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 );
    }
}
```

В этом фрагменте `frame` — переменная. В строке 5 она заполняется новым экземпляром класса `JFrame`. Затем тот же экземпляр повторно используется

в строке 7 для добавления надписи. В строке 8 эта переменная снова нужна нам, чтобы установить размер окна, а в строке 9 — чтобы перевести окно в видимое состояние. Повторное использование данных — это та область, в которой переменные по-настоящему проявляют себя.

В строке 6 есть константа `JLabel.CENTER`. Константы содержат значения, которые остаются неизменными во время выполнения программы. Казалось бы, зачем хранить таким образом те значения, которые не будут изменяться? Почему бы просто не вписывать их в код каждый раз, когда они понадобятся? Дело в том, что автор кода может выбирать имена для всех констант, что сразу приносит пользу: значения дополняются содержательными описаниями. Возможно, смысл имени `JLabel.CENTER` все еще не совсем очевиден, но слово `CENTER` по крайней мере намекает на то, что здесь происходит.

Кроме того, именованные константы упрощают внесение изменений. Например, если в вашем коде указано предельное количество единиц какого-то используемого ресурса, то скорректировать этот предел будет намного проще, если для этого достаточно изменить инициализированное значение константы. Если бы для этой же цели использовался числовой литерал, например `5`, то вам пришлось бы выискивать во всех файлах Java все вхождения `5` и изменять их каждый раз, когда конкретный экземпляр `5` действительно относится к ограничению соответствующего ресурса. Такой ручной поиск с заменой крайне ненадежен, не говоря уже об однообразии работы.

Типы и исходные значения переменных и констант более подробно описываются далее, в следующем разделе. Как обычно, не стесняйтесь использовать `jshell`, чтобы найти и исследовать некоторые из этих подробностей самостоятельно! Впрочем, помните, что из-за ограничений интерпретатора вы не сможете объявлять собственные константы верхнего уровня в `jshell`. Но вы можете использовать константы, определенные для классов (такие, как упомянутая выше `JLabel.CENTER`), а также определять константы в ваших собственных классах, вводимых в `jshell`. Класс `Math` содержит множество разных математических функций и константу, представляющую число π . Попробуем вычислить и сохранить площадь круга в переменной, а затем продемонстрировать, что повторное присваивание констант не работает.

```
jshell> double radius = 42.0;
radius ==> 42.0

jshell> Math.PI
$2 ==> 3.141592653589793

jshell> Math.PI = 3;
| Error:
| cannot assign a value to final variable PI
| Math.PI = 3;
| ^-----^
```

```
jshell> double area = Math.PI * radius * radius;  
area ==> 5541.769440932396
```

```
jshell> radius = 6;  
radius ==> 6.0
```

```
jshell> area = Math.PI * radius * radius;  
area ==> 113.09733552923255
```

```
jshell> area  
area ==> 113.09733552923255
```

Обратите внимание на ошибку компилятора при попытке присвоить числу π значение 3. Также обратите внимание на то, что значения `radius` и `area` **можно** менять после объявления и инициализации. Тем не менее в любой момент времени переменная может хранить только одно значение. В переменной `area` остается только последний вычисленный результат.

Типы

Система типов в языке программирования описывает, как его элементы данных (только что упоминавшиеся переменные и константы) связываются со своими блоками памяти и как они связываются друг с другом. В языке со статической типизацией (например, C или C++) тип элемента данных представляет собой простой неизменяемый атрибут, который часто соответствует некоторому аппаратному понятию, например регистру или значению указателя. В более динамических языках (таких, как Smalltalk или Lisp) переменные могут обозначать произвольные элементы и даже изменять свои типы на протяжении своего срока жизни. На проверку того, что происходит в этих языках во время выполнения кода, расходуются значительные ресурсы. Языки сценариев, такие как Perl, достигают простоты использования благодаря радикально упрощенной системе типов, в которой переменные могут хранить только некоторые элементы данных, а значения объединяются в общее представление (например, строки).

В Java сочетаются многие лучшие свойства языков как со статической, так и с динамической типизацией. Как и в языках со статической типизацией, каждая переменная и каждый программный элемент в Java имеют тип, известный на стадии компиляции, поэтому исполнительной системе обычно не нужно проверять корректность присваиваний между типами во время выполнения кода. Кроме того, в отличие от традиционных C и C++, Java контролирует информацию об объектах во время выполнения и использует ее для реализации полноценного динамического поведения. Код Java может загружать новые типы во время выполнения и использовать их объектно-ориентированными способами, допуская

преобразования типов и полноценный полиморфизм (расширение типов). Код Java также может анализировать свои типы во время выполнения, что делает возможным очень сложное поведение приложений, например интерпретаторов, способных динамически взаимодействовать со скомпилированными программами.

Типы данных Java делятся на две категории. *Примитивные типы* (*primitive types*) представляют простые значения, обладающие в языке встроенной функциональностью, такие как числа, логические операторы и символы текста. *Ссылочные типы* (*reference types*), или типы классов, включают объекты и массивы; они называются ссылочными типами, потому что они «ссылаются» на большие по объему данные, определяя их «по ссылкам» (вскоре мы объясним, что это означает). *Обобщенные типы* (*generic types*) и методы определяют объекты разных типов и работают с ними, обеспечивая безопасность типов при компиляции. Например, `List<String>` — это список `List`, который может содержать только элементы `String`. Обобщенные типы тоже являются ссылочными типами; в главе 7 мы рассмотрим много примеров такого рода.

Примитивные типы

Числа, символы и логические значения (*boolean values*) относятся к фундаментальным элементам Java. В отличие от других объектно-ориентированных языков (возможно, более чистых), они не являются объектами. Но для ситуаций, в которых необходимо работать с примитивным типом как с объектом, Java предоставляет *классы-обертки* (*wrapper classes*); о них мы расскажем далее. Главное преимущество обработки примитивных типов как специальных заключается в том, что компилятору и исполнительной системе Java проще оптимизировать их реализацию. Примитивные значения и вычисления по-прежнему могут отображаться на аппаратные элементы компьютера, как всегда было в низкоуровневых языках. Если вы будете работать с платформенными библиотеками, используя JNI (Java Native Interface) для взаимодействия с другими языками или службами, то эти примитивные типы займут важное место в вашем коде.

Важная особенность портируемости Java — точное определение примитивных типов. Например, вам никогда не придется беспокоиться о размере переменных типа `int` на конкретной платформе; этот тип всегда будет 32-разрядным числом со знаком, которое представлено в дополнительном коде. Размер числовой переменной определяет, насколько большое (или насколько точное) значение в ней можно сохранить. Например, тип `byte` предназначен для малых чисел в диапазоне от -128 до 127 , а тип `int` обеспечивает большинство потребностей в работе с числами, позволяя хранить значения в диапазоне ± 2 миллиарда (приблизительно). В табл. 4.2 перечислены все примитивные типы Java.

Таблица 4.2. Примитивные типы данных Java

Тип	Определение	Приблизительный диапазон или точность
boolean	Логическое значение	true или false
char	16-разрядный символ Unicode	64 К символов
byte	8-разрядное целое со знаком в дополнительном коде	от -128 до 127
short	16-разрядное целое со знаком в дополнительном коде	от -32 768 до 32 767
int	32-разрядное целое со знаком в дополнительном коде	от -2,1e9 до 2,1e9
long	64-разрядное целое со знаком в дополнительном коде	от -9,2e18 до 9,2e18
float	32-разрядное число с плавающей точкой в формате IEEE 754	6-7 значимых цифр в дробной части
double	64-разрядное число с плавающей точкой в формате IEEE 754	15 значимых цифр в дробной части



Читатели с опытом программирования на C заметят, что примитивные типы выглядят как идеализация скалярных типов C на 32-разрядной машине, и это правда. Именно так они и должны выглядеть. 16-разрядные символы были обусловлены стандартом Unicode, а произвольные указатели были исключены по другим причинам. Но в целом синтаксис и семантика примитивных типов Java происходят от C.

Но зачем вообще нужны разные размеры? И снова все возвращается к эффективности и оптимизации. Количество голов в футбольном матче редко выходит за пределы однозначных чисел, то есть его можно хранить в переменной `byte`. Для того, чтобы обозначить количество болельщиков, смотрящих матч, потребуется что-то побольше. А общая сумма денег, потраченных всеми болельщиками во всех футбольных матчах чемпионата мира, займет еще больше памяти. Выбирая правильный размер, вы предоставляете компилятору максимальные возможности для оптимизации кода, что позволяет ускорить работу приложения и (или) потреблять меньше системных ресурсов.

Если вам нужны очень большие числа, которые не поддерживаются примитивными типами, обратите внимание на классы `BigInteger` и `BigDecimal` из пакета `java.Math`. Эти классы обеспечивают почти бесконечные размеры и точность. Некоторые научные или криптографические приложения, требующие хранения и обработки очень больших (или очень малых) чисел, отдают предпочтение точности перед быстродействием. В книге эти классы описываться не будут, но вы на всякий случай запомните их для исследований в ненастный день.

Точность чисел с плавающей запятой

В Java все операции с плавающей запятой (которая в международной терминологии называется *плавающей точкой*: *floating point*) определяются международной спецификацией IEEE 754. Это означает, что результаты вычислений с плавающей точкой, как правило, всегда одинаковы на всех платформах Java. Тем не менее Java позволяет выполнять вычисления с повышенной точностью на тех платформах, которые поддерживают такую возможность. Это может создать крайне малые и трудные для понимания различия в результатах операций с высокой точностью. В большинстве приложений вы их никогда не заметите, но если вы хотите гарантировать, что приложение выдает идеально совпадающие результаты на всех платформах, используйте специальное ключевое слово `strictfp` в качестве модификатора для класса, содержащего операции с плавающей точкой (классы рассматриваются в следующей главе). Тогда компилятор запретит те виды оптимизации, которые зависят от платформы.

Объявление и инициализация переменных

Переменные объявляются внутри методов и классов путем указания имени типа, за которым следует одно или несколько имен переменных, разделенных запятыми. Пример:

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

Переменные можно инициализировать выражением соответствующего типа при объявлении:

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

Если переменные, объявленные в составе класса, не были инициализированы (см. главу 5), то им присваиваются значения по умолчанию. В таком случае переменным числовых типов по умолчанию присваивается ноль, символам присваивается `null`-символ (`\0`), а логическим переменным — значение `false`. (У ссылочных типов также имеется значение по умолчанию `null`, но об этом — далее, в разделе «Ссылочные типы» на с. 124.) Но локальные переменные, объявленные внутри метода и существующие только во время вызова метода, должны быть явно инициализированы перед использованием. Как вы увидите, компилятор заставляет вас соблюдать это правило, поэтому нет риска забыть о нем.

Целочисленные литералы

Целочисленные литералы могут записываться в двоичной системе (основание 2), а также в восьмеричной (основание 8), десятичной (основание 10) или шест-

надцатеричной (основание 16). Двоичные, восьмеричные и шестнадцатеричные данные в основном используются при работе с низкоуровневой информацией из файлов или сетевых данных. Они представляют собой удобные группы отдельных битов: из 1, 3 и 4 битов соответственно. У десятичных значений такого соответствия нет, но, как правило, они намного удобнее для человека при работе с числовой информацией. Десятичное целое число обозначается последовательностью цифр, начинающейся с одного из символов в диапазоне от 1 до 9:

```
int i = 1230;
```

Двоичное число обозначается начальными символами `0b` или `0B`, за которыми следует последовательность нулей и единиц:

```
int i = 0b01001011; // i = 75 (в десятичной записи)
```

Восьмеричные числа можно отличить от десятичных по начальному нулю:

```
int i = 01230; // i = 664 (в десятичной записи)
```

Шестнадцатеричное число состоит из начальных символов `0x` (или `0X`), за которыми следует последовательность цифр и символов `a–f` (или `A–F`), означающих десятичные числа от 10 до 15:

```
int i = 0xFFFF; // i = 65535 (в десятичной записи)
```

Целочисленные литералы по умолчанию имеют тип `int`, если только они не завершаются суффиксом `L`, который означает, что они представляют значение типа `long`:

```
long l = 13L;
long l = 13; // Эквивалент: число 13 преобразуется из типа int
long l = 40123456789L;
long l = 40123456789; // Ошибка: слишком большое число для int
                      // без преобразования
```

Буква `l` нижнего регистра тоже допустима, но лучше ее не использовать, потому что она слишком похожа на цифру `1`.

Когда числовой тип используется в присваивании или в выражении вместе с типом большего размера (с большим диапазоном), он может быть автоматически *повышен* до большего типа. Во второй строке приведенного выше примера число `13` (целочисленный литерал) по умолчанию имеет тип `int`, но он повышается до типа `long` для присваивания переменной `long`. Другие числовые операции и операции сравнения тоже приводят к подобным арифметическим повышениям, как и математические выражения, в которых задействовано несколько типов. Например, при умножении значения `byte` на значение `int` компилятор сначала повышает `byte` до `int`:

```
byte b = 42;
int i = 43;
int result = b * i; // b повышается до int перед умножением
```

Но обратная ситуация невозможна: числовое значение не может быть присвоено типу с меньшим диапазоном без явного преобразования:

```
int i = 13;
byte b = i; // Ошибка компиляции, требуется явное преобразование
byte b = (byte) i; // ОК
```

Преобразования типов с плавающей точкой в целочисленные типы всегда должны быть явными во избежание возможной потери точности.

Начиная с Java 7, к числовым литералам можно применять простейшее форматирование, разделяя цифры символом подчеркивания: `_`. Таким образом, если вы работаете с длинными строками цифр, их можно разбивать на легко читаемые группы как в следующем примере:

```
int RICHARD_NIXONS_SSN = 567_68_0515;
int for_no_reason = 1__2__3;
int JAVA_ID = 0xCAFE_BABE;
long grandTotal = 40_123_456_789L;
```

Символы подчеркивания могут находиться только между цифрами, но не в начале числа, не в конце числа и не рядом с суффиксом `L` — признаком типа `long`. Попробуйте ввести несколько больших чисел в `jshell`. Обратите внимание: при попытке сохранить тип `long` без суффикса возникает ошибка. Как видите, форматирование нужно только для вашего удобства. Оно не сохраняется во внутреннем представлении; в переменной или константе хранится только само значение:

```
jshell> long m = 41234567890;
| Error:
| integer number too large
| long m = 41234567890;
|      ^
jshell> long m = 40123456789L;
m ==> 40123456789

jshell> long grandTotal = 40_123_456_789L;
grandTotal ==> 40123456789
```

Поэкспериментируйте с другими примерами. Так вы сможете понять, какой вариант лучше всего подходит лично вам. Также вы получите представление о том, какие преобразования и повышения возможны или обязательны. Ничто не закрепит все эти нюансы в вашей памяти так хорошо, как немедленная обратная связь!

Литералы с плавающей точкой

Значения с плавающей точкой могут задаваться в обычной записи или в научной записи. Литералы с плавающей точкой обычно относятся к типу `double`, если только они не завершаются суффиксом `f` или `F`, который означает, что они представляют значения типа `float`. Как и с целочисленными литералами, начиная с Java 7, можно использовать символы `_` для форматирования чисел с плавающей точкой — но только между цифрами, а не в начале числа, не в конце числа и не рядом с точкой или суффиксом `F`.

```
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;
```

Символьные литералы

Значение символьного литерала задается либо символом, заключенным в одинарные кавычки, либо экранированной последовательностью (escape-последовательностью) ASCII или Unicode:

```
char a = 'a';
char newline = '\n';
char smiley = '\u263a';
```

Ссылочные типы

Java — это объектно-ориентированный язык, поэтому вы можете создавать из простых примитивов новые, более сложные типы данных при помощи классов. Каждый класс становится новым типом данных в языке. Например, если вы создаете новый класс с именем `Foo`, вы тем самым создаете новый тип данных с таким же именем `Foo`. Тип элемента определяет, как он используется и где он может присваиваться. Как и в случае с примитивами, элемент типа `Foo` в общем случае может быть присвоен переменной типа `Foo` или передан в аргументе методу, получающему значение типа `Foo`.

Тип — это не просто атрибут. Классы могут быть связаны разными отношениями с другими классами, как и представляемые ими типы. Все классы в Java существуют в рамках иерархии «родитель — потомок», в которой *субкласс*, или *подкласс* (то есть дочерний класс), является особым частным случаем своего *суперкласса* (то есть родительского класса). Соответствующие типы обладают той же связью: тип субкласса считается подтипом суперкласса. Поскольку субклассы наследуют всю функциональность своих суперклассов, объект дочернего типа отчасти эквивалентен родительскому типу или является его *расширением*. Объект субкласса может использоваться вместо объекта суперкласса. Например, если вы создали новый

класс `Cat`, являющийся расширением класса `Animal`, то новый тип `Cat` будет под-типом `Animal`. После этого объекты типа `Cat` можно использовать везде, где может использоваться объект типа `Animal`; говорят, что объект типа `Cat` *совместим по присваиванию* с переменной типа `Animal`. Этот принцип, называемый *полиморфизмом подтипов*, является одной из главных особенностей объектно-ориентированных языков. Классы и объекты детально рассматриваются в главе 5.

Все примитивные типы в Java используются и передаются «по значению». Иначе говоря, когда примитивное значение (например, `int`) присваивается переменной или передается в аргументе методу, Java выполняет копирование этого значения. С другой стороны, все обращения к ссылочным типам (к типам классов) происходят «по ссылкам». *Ссылка (reference)* — это именованный идентификатор объекта. В переменной любого ссылочного типа хранится указатель на адрес в памяти, где находится конкретный объект ее типа (или ее подтипа). Когда ссылка присваивается переменной или передается в аргументе методу, Java выполняет копирование этой ссылки, а не самого объекта, на который она указывает. Ссылки похожи на указатели в C и C++, но с тем важным отличием, что все типы ссылок находятся под строгим контролем. В Java невозможно явным образом назначить или изменить значение переменной ссылочного типа (указатель на адрес в памяти). Такая переменная получает свое значение автоматически, только при присваивании ей конкретного объекта.

Рассмотрим пример. Объявим переменную типа `Foo` с именем `myFoo` и присвоим ей подходящий по типу объект¹:

```
Foo myFoo = new Foo();
Foo anotherFoo = myFoo;
```

Здесь `myFoo` — переменная ссылочного типа, содержащая ссылку на созданный объект `Foo`. (Пока не будем отвлекаться на подробное описание создания объектов; эта тема рассматривается в главе 5.) Объявим вторую переменную `anotherFoo` типа `Foo` и присвоим ей тот же объект. Теперь в программе существуют две идентичные ссылки: `myFoo` и `anotherFoo`, но только один экземпляр `Foo`. Если изменить что-либо в состоянии самого объекта `Foo`, то эти изменения будут видны по любой из двух ссылок. Чтобы «заглянуть за кулисы» и понять, что происходит при работе со ссылками, выполните в `jshell` следующий код:

```
jshell> class Foo {}
| created class Foo

jshell> Foo myFoo = new Foo()
myFoo ==> Foo@21213b92
```

¹ Эквивалентный код C++ выглядит так:

```
Foo& myFoo = *(new Foo());
Foo& anotherFoo = myFoo;
```

```
jshell> Foo anotherFoo = myFoo  
anotherFoo ==> Foo@21213b92
```

```
jshell> Foo notMyFoo = new Foo()  
notMyFoo ==> Foo@66480dd7
```

Обратите внимание на результаты создания и присваивания (конечно, на вашем компьютере значения указателей будут другими). Вы видите, что переменная ссылочного типа определяется значением указателя (21213b92, справа от @) и типом (Foo, слева от @). При создании нового объекта типа Foo вы получите новое значение указателя. Переменные `myFoo` и `anotherFoo` указывают на один и тот же объект. Переменная `notMyFoo` указывает на второй, отдельный объект того же типа.

Автоматическое определение типов

В современных версиях Java постоянно совершенствуется способность компилятора автоматически определять типы переменных во многих ситуациях. Используйте ключевое слово `var` в сочетании с объявлением и инициализацией переменной, чтобы поручить компилятору определить правильный тип:

```
jshell> class Foo2 {}  
| created class Foo2
```

```
jshell> Foo2 myFoo2 = new Foo2()  
myFoo2 ==> Foo2@728938a9
```

```
jshell> var myFoo3 = new Foo2()  
myFoo3 ==> Foo2@6433a2
```

Обратите внимание на вывод `myFoo3` в `jshell`. Хотя мы не указали тип явно, как было сделано для `Foo2`, компилятор легко определяет, какой тип следует использовать, и мы получаем объект типа `Foo2`.

Передача ссылок

Ссылки на объекты передаются методам по одной схеме. В нашем случае аргументы `myFoo` и `anotherFoo` будут эквивалентными:

```
myMethod( myFoo );
```

Теперь надо упомянуть о важном отличии, которое иногда вызывает недоразумения: сама ссылка является значением, и именно это значение копируется при присваивании переменной или при передаче в вызове метода. В приведенном выше примере аргумент, полученный методом (с точки зрения мето-

да — локальная переменная) в действительности является третьей ссылкой на объект `foo`, в дополнение к `myFoo` и `anotherFoo`. Метод может изменить состояние объекта `foo` по этой ссылке (вызывая его методы или изменяя его переменные), но не может изменить смысл ссылки на `myFoo` с вызывающей стороны. Иначе говоря, метод не может изменить ссылку `myFoo` на стороне вызова и заставить ее указывать на какой-то другой объект `foo`. Метод может изменить только свою ссылку. Это станет более очевидным, когда мы будем рассказывать о методах. В этом отношении Java отличается от C++. Если вы захотите изменить ссылку на объект на стороне вызова в Java, для этого потребуется дополнительный уровень косвенного обращения. То есть сторона вызова должна будет «обернуть» ссылку в другой объект, чтобы обе стороны могли использовать общую ссылку на него.

Ссылочные типы всегда содержат указатели на объекты (или содержат `null`), а объекты всегда определяются классами. По аналогии с платформенными типами всем переменным экземпляров или классов, которые не инициализируются явно, при объявлении присваивается значение по умолчанию `null`. Как и платформенные типы, локальные переменные ссылочных типов **не** инициализируются по умолчанию, поэтому вы должны явно присвоить им значения перед использованием. Впрочем, две особые разновидности ссылочных типов — массивы и интерфейсы — несколько иначе задают типы объектов, на которые они ссылаются.

Массивы в Java занимают особое место в системе типов. Массив — это особая разновидность объектов; он автоматически создается для хранения групп объектов другого типа (который называется *базовым типом*). При объявлении ссылки с типом массива неявно создается новый тип класса, который служит контейнером для своего базового типа, как вы увидите далее в этой главе.

С интерфейсами дело обстоит сложнее. Интерфейс определяет набор методов и назначает ему соответствующий тип. К объекту, реализующему методы интерфейса, можно обращаться как по типу интерфейса, так и по его собственному типу. Переменные и аргументы методов могут объявляться как относящиеся к типу интерфейса (как и любые другие типы классов) и им могут быть присвоены любые объекты, реализующие интерфейс. В результате система типов становится более гибкой: она позволяет Java выходить за границы иерархии классов и создавать объекты, которые фактически обладают многими типами. Интерфейсы будут рассмотрены в следующей главе.

Обобщенные типы (или *параметризованные типы*), как упоминалось ранее, являются расширением синтаксиса классов Java, которое реализует дополнительный уровень абстракции при работе классов с другими типами Java. Этот механизм позволяет изменять специализацию класса без изменения кода исходного класса. Обобщения подробно рассматриваются в главе 7.

Несколько слов о строках

Строки в Java являются объектами; следовательно, они относятся к ссылочным типам. Тем не менее объекты `String` получают от компилятора дополнительную поддержку, благодаря которой они больше похожи на примитивные типы. Литеральные строковые значения в исходном коде Java преобразуются в объекты `String` компилятором. Их можно использовать напрямую, передавать в аргументах методам или присваивать переменным типа `String`:

```
System.out.println( "Hello, World..." );
String s = "I am the walrus...";
String t = "John said: \"I am the walrus...\"";
```

Оператор `+` в Java *перегружается*, чтобы выполнять не только обычное числовое сложение, но и конкатенацию. Оператор `+` и родственный ему `+=` являются единственными перегружаемыми операторами в Java:

```
String quote = "Four score and " + "seven years ago,";
String more = quote + " our" + " fathers" + " brought...";
```

Здесь Java составляет объект `String` из строк, объединенных конкатенацией, и предоставляет его как результат выражения. Класс `String` и работа с текстом подробно рассматриваются в главе 8.

Команды и выражения

Команды Java размещаются внутри методов и классов; они описывают все выполняемые в программе операции. Объявления переменных и присваивания (вроде приведенных в предыдущем разделе) являются командами, как и базовые структуры языка вроде условных конструкций `if / else` и циклов. (Подробнее об этих структурах рассказано далее в этой главе.)

```
int size = 5;
if ( size > 10 )
    doSomething();
for ( int x = 0; x < size; x++ ) { ... }
```

Выражения предоставляют значения; выражение вычисляется для получения результата, который может использоваться как часть другого выражения или команды. Примеры выражений — вызовы методов, создание объектов и, конечно, математические выражения.

```
new Object()
Math.sin( 3.1415 )
42 * 64
```


Один из основополагающих принципов Java — простота и последовательность. При отсутствии других ограничений вычисление и инициализация в Java всегда выполняются в порядке их следования в коде: слева направо, сверху вниз. Вы увидите, как это правило используется для вычисления выражений присваивания, для вызовов методов, для индексирования массивов и т. д. В некоторых языках порядок вычисления бывает более сложным или даже зависящим от реализации. Java исключает этот потенциальный риск, просто и точно определяя последовательность обработки кода. Впрочем, это не означает, что вам надо писать малопонятные и запутанные команды. Сложные зависимости от порядка вычисления выражений — плохой стиль программирования, даже если код работает. Такой код хуже читается и в него сложнее вносить изменения.

Команды

В любой программе все самое интересное делают команды. Они помогают реализовать алгоритмы, упомянутые в начале главы. Собственно, они не просто помогают, а становятся основными инструментами для решения задач; каждый шаг алгоритма соответствует одной или нескольким командам. Команды обычно выполняют одну из четырех операций: ввод данных для присваивания переменным; вывод данных (на терминал, в `JLabel` и т. д.); принятие решений относительно того, какие команды должны быть выполнены далее; повторение одной или нескольких других команд. Рассмотрим примеры всех этих категорий в Java.

Команды и выражения в Java размещаются в *блоках кода*. С точки зрения синтаксиса блок представляет собой серию команд, заключенных между открывающей (`{`) и закрывающей (`}`) фигурными скобками. В блоке могут находиться команды объявления переменных, а также многие другие команды и выражения, упоминавшиеся ранее:

```
{
    int size = 5;
    setName( "Max" );
    ...
}
```

Методы в Java напоминают функции языка C. В каком-то смысле это обычные блоки, которые могут получать параметры и вызываться по имени. Например, мы могли бы написать метод с именем `setUpDog()`:

```
setUpDog( String name ) {
    int size = 5;
    setName( name );
    ...
}
```

Видимость объявленной в блоке переменной ограничивается этим блоком, то есть переменная не видна за пределами ближайшей пары фигурных скобок:

```
{
    int i = 5;
}
i = 6; // Ошибка компиляции, переменная i не существует
```

Таким образом, блоки удобны для произвольной группировки команд и переменных. Тем не менее самое распространенное применение блоков — определение групп команд для использования в условных или итеративных командах.

Условные команды if / else

Принятие решений — одна из ключевых концепций программирования. «Если этот файл существует...» или «если у пользователя есть подключение Wi-Fi...» — подобные решения принимаются компьютерными программами постоянно. Условие if / else может определяться так:

```
if ( условие )
    команда;
else
    команда;
```

Весь этот пример является командой, и он может быть вложен в другую команду if / else. Конструкция if может существовать в двух разных формах: однострочной и блочной. Блочная форма выглядит так:

```
if ( условие ) {
    [ команда; ]
    [ команда; ]
    [ ... ]
} else {
    [ команда; ]
    [ команда; ]
    [ ... ]
}
```

Условие — это логическое выражение (boolean expression). Оно может иметь значение true или false либо может быть таким выражением, при вычислении которого будет получено одно из этих значений. Например, `i == 0` — это логическое выражение, которое проверяет, содержит ли переменная `i` значение 0.

Во второй форме команды заключены в блоки, а все команды в блоке выполняются только при выборе соответствующей ветви: if или else. Любые переменные, объявленные в каждом блоке, видимы только в командах этого блока.

Кроме команд `if / else`, многие другие команды Java тоже управляют последовательностью выполнения. Они работают примерно так же, как и их аналоги в других языках.

Команды `switch`

Во многих языках поддерживается условная конструкция «выбор одного из многих», обычно называемая командой `switch` или `case`. Команда `switch` проверяет заданную переменную или выражение на возможное совпадение с несколькими перечисленными вариантами. Предпочтение отдается первому найденному совпадению, поэтому важен порядок перечисления. И мы неспроста говорим, что совпадение «возможное», так как значение может не совпасть ни с одним из вариантов, перечисленных в `switch`. В таком случае ничего не происходит.

Самая распространенная форма команды `switch` получает целое число (или аргумент числового типа, который может быть автоматически повышен до целочисленного типа), строку или перечисление (см. далее) — и выбирает между несколькими ветвями `case` с альтернативами-константами¹:

```
switch ( выражение )
{
    case выражениеКонстанта :
        команда;
    [ case выражениеКонстанта :
        команда; ]
    ...
    [ default :
        команда; ]
}
```

Выражения `case` для всех ветвей должны давать разные целочисленные константы или строки на стадии компиляции. Строки сравниваются методом `equals()` класса `String`, который будет подробнее рассмотрен в главе 8. Команда в необязательной секции `default` выполняется, если ни одно из условий не совпало. При выполнении команда `switch` находит ветвь, соответствующую условному выражению (или ветвь `default`), и выполняет команду. Тем не менее это еще не все. Как ни странно, команда `switch` затем продолжает выполнять другие ветви (уже после ветви с совпадением), пока не дойдет до конца блока `switch` или до специальной команды `break`. Пара примеров:

```
int value = 2;
switch( value ) {
    case 1:
        System.out.println( 1 );
    case 2:
```

¹ Поддержка строк в командах `switch` появилась в Java 7.

```
        System.out.println( 2 );
    case 3:
        System.out.println( 3 );
}
// Выводятся 2 и 3
```

Чаще всего для завершения каждой ветви используется команда `break`:

```
int retValue = checkStatus();
switch ( retVal )
{
    case MyClass.GOOD :
        // Хороший вариант
        break;
    case MyClass.BAD :
        // Плохой вариант
        break;
    default :
        // Ни то ни другое
        break;
}
```

В этом примере выполняется только одна ветвь — `GOOD`, `BAD` или `default`. «Сквозная» передача управления в `switch` нужна, когда вы хотите обработать несколько возможных значений одной командой без использования лишних конструкций `if / else`:

```
int value = getSize();
String size = "Unknown";

switch( value ) {
    case MINISCULE:
    case TEENYWEENIE:
    case SMALL:
        size = "Small";
        break;
    case MEDIUM:
        size = "Medium";
        break;
    case LARGE:
    case EXTRALARGE:
        size = "Large";
        break;
}

System.out.println("Your size is: " + size);
```

Здесь шесть возможных значений группируются на три случая. И теперь возможность группировки может использоваться напрямую в выражениях: в Java 12 появилась предварительная версия *выражений switch*. Например, приведенный

выше код можно модифицировать, создав переменную `size`, представляющую размер:

```
int value = getSize();
String size = switch( value ) {
    case MINISCULE:
    case TEENYWEENIE:
    case SMALL:
        break "Small";
    case MEDIUM:
        break "Medium";
    case LARGE:
    case EXTRALARGE:
        break "Large";
}
System.out.println("Your size is: " + size);
```

Обратите внимание: в этом случае команда `break` используется со значением. Новый синтаксис также можно использовать в командах `switch`, чтобы код стал компактнее и понятнее:

```
int value = getSize();
String size = switch( value ) {
    case MINISCULE, TEENYWEENIE, SMALL -> "Small";
    case MEDIUM -> "Medium";
    case LARGE, EXTRALARGE -> "Large";
}
System.out.println("Your size is: " + size);
```

Эти выражения появились в языке относительно недавно (в Java 12 для работы с ними приходилось компилировать программу с флагом `--enable-preview`), так что они все еще редко встречаются в интернете. Но если команда `switch` вас заинтересует, вы наверняка найдете хорошие примеры, объясняющие мощь выражений `switch`.

Циклы `do / while`

Другая важная концепция, относящаяся к выбору команды, которая должна выполняться следующей, — *повторение*. Компьютеры отлично справляются с многократным выполнением одних и тех же действий. Блоки кода повторяются в *циклах*. В Java существуют две основные разновидности циклов. Команды `do` и `while` выполняются, пока логическое выражение возвращает значение `true`:

```
while ( условие )
    команда;

do
    команда;
while ( условие );
```

Цикл `while` идеально подходит для ожидания некоторого внешнего условия, например получения электронной почты:

```
while( mailQueue.isEmpty() )  
    wait();
```

Конечно, метод ожидания `wait()` должен иметь ограничение (обычно ограничение по времени, например ожидание в течение секунды), чтобы он завершился и позволил циклу выполниться снова. Но когда в очереди появится электронная почта, обработать нужно будет все поступившие сообщения, а не одно. И снова цикл `while` идеально подходит для этой задачи:

```
while( !mailQueue.isEmpty() ) {  
    EmailMessage message = mailQueue.takeNextMessage();  
    String from = message.getFromAddress();  
    System.out.println("Processing message from " + from);  
    message.doSomethingUseful();  
}
```

В этом маленьком фрагменте логический оператор `!` инвертирует результат предыдущей проверки. То есть работа должна продолжаться в том случае, если в очереди что-то есть (если **не** выполняется условие, что очередь пустая). Обратите внимание, что тело цикла состоит из нескольких команд, поэтому мы заключили его в фигурные скобки. Внутри этих скобок мы извлекаем из очереди сообщение и сохраняем его в локальной переменной `message`. Затем мы выполняем с переменной `message` некоторые действия, после чего цикл возвращается к проверке условия пустой очереди. Если очередь не пустая, то весь процесс повторяется, начиная со следующего доступного сообщения.

В отличие от цикла `while` и цикла `for` (см. далее), которые начинаются с проверки своих условий, цикл `do-while` (часто называемый циклом `do`) всегда выполняет свое тело хотя бы один раз. Классический пример — проверка ввода от пользователя, например, на веб-сайте. Вы знаете, что вам нужно получить некоторые данные, поэтому запрашиваете их в теле цикла. В условии цикла можно проверять вводимые данные на наличие ошибок. При обнаружении ошибки цикл перезапускается, а данные запрашиваются снова. Этот процесс повторяется, пока запрос не будет обработан без ошибок, что обеспечивает корректность полученных данных.

Цикл `for`

Самая общая форма цикла `for` является наследием языка C:

```
for ( инициализация; условие; приращение )  
    команда;
```

В секции *инициализация* могут объявляться или инициализироваться переменные, видимость которых ограничивается областью видимости команды.

Цикл `for` затем начинает серию проверок, в которых сначала проверяется *условие*, и если оно истинно, выполняется тело команды (или блок). После каждого прохода тела цикла выполняются выражения из секции *приращение*, которая дает возможность обновить переменные перед следующей итерацией:

```
for ( int i = 0; i < 100; i++ ) {  
    System.out.println( i );  
    int j = i;  
    ...  
}
```

В этом примере цикл выполняется 100 раз и выводит значения от 0 до 99. Обратите внимание на то, что переменная `j` локальна для блока (видна только в командах внутри этого блока) и становится недоступной в коде после цикла `for`. Если условие цикла `for` возвращает `false` при первой проверке, то тело цикла и секция *приращение* выполняться не будут.

В секциях *инициализация* и *приращение* цикла `for` можно использовать несколько выражений, разделенных запятыми. Пример:

```
for ( int i = 0, j = 10; i < j; i++, j-- ) {  
    System.out.println(i + " < " + j);  
    ...  
}
```

Также в блоке инициализации можно инициализировать существующие переменные, объявленные за пределами области видимости цикла `for`. Например, это можно сделать, если вы хотите использовать конечное значение переменной цикла в другом месте программы. Обычно не рекомендуется так поступать, поскольку велик риск ошибок и ваш код может стать трудным для понимания. Тем не менее такая возможность существует, поэтому вы можете столкнуться с ситуацией, в которой эта логика работы покажется вам разумной.

```
int x;  
for( x = 0; hasMoreValue(); x++ ) {  
    getNextValue();  
}  
// Значение x остается доступным  
System.out.println( x );
```

Расширенный цикл `for`

Расширенный цикл `for` в Java похож на команду `foreach` в некоторых других языках. Эта команда перебирает серию значений из массива или коллекцию другого типа:

```
for ( объявлениеПеременной : значения )  
    команда;
```

Расширенный цикл `for` может использоваться для перебора массивов других типов, а также любых объектов Java, реализующих интерфейс `java.lang.Iterable`. К этой категории относится большинство классов из API коллекций Java. Массивы будут рассматриваться в этой и следующей главах; в главе 7 рассматриваются коллекции Java. 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 );
```

Подчеркнем: в этом примере не обсуждаются массивы, класс `List` или специальный синтаксис. Здесь демонстрируется лишь расширенный цикл `for` с перебором массива целых чисел и списка строковых значений. Во втором случае `List` реализует интерфейс `Iterable`, а следовательно, может использоваться в расширенном цикле `for`.

Команды `break` / `continue`

Команда `break` и родственная ей команда `continue` могут использоваться для ускоренного выхода из цикла или условной команды. Команда `break` заставляет Java прервать текущую команду цикла (или `switch`) и продолжить выполнение после нее. В следующем примере цикл `while` продолжается бесконечно, пока метод `condition()` не вернет `true`; в этом случае срабатывает команда `break`, которая прерывает цикл и передает управление в точку с комментарием «после `while`»:

```
while( true ) {
    if ( condition() )
        break;
}
// После while
```

Команда `continue` заставляет циклы `for` и `while` перейти к следующей итерации, передавая управление в точку проверки условия. Следующий код выводит числа от 0 до 99, пропуская 33:

```
for( int i=0; i < 100; i++ ) {
    if ( i == 33 )
        continue;
    System.out.println( i );
}
```


Команды `break` и `continue` похожи на одноименные команды языка C, но их формы в Java могут получать в аргументе метку и передавать управление через несколько уровней к помеченной точке кода. Такое использование нетипично для современного программирования на Java, но может пригодиться в особых случаях. Общая схема выглядит так:

```
labelOne:
    while ( условие ) {
        ...
        labelTwo:
            while ( условие ) {
                ...
                // Точка для команды break или continue
            }
            // После labelTwo
        }
    }
// После labelOne
```

Команды-контейнеры (блоки, условные команды, циклы и т. д.) могут помечаться метками, такими как `labelOne` и `labelTwo`. Если в данном примере поместить в указанную точку команду `break` или `continue` без аргумента, то мы получим такой же эффект, как в предыдущем примере: команда `break` приводит к продолжению выполнения в точке с комментарием «После `labelTwo`»; команда `continue` немедленно заставляет цикл `labelTwo` вернуться к проверке условия.

Если в указанной точке находится команда `break labelTwo`, то она делает то же, что и обычная команда `break`, но команда `break labelOne` переходит на два уровня вверх и возобновляет выполнение в точке с комментарием «После `labelOne`». Аналогичным образом команда `continue labelTwo` работает как обычная команда `continue`, но команда `continue labelOne` возвращается к проверке цикла `labelOne`. Многоуровневые команды `break` и `continue` устраняют главное оправдание для столь порицаемой команды `goto` в языках C и C++¹.

В Java есть и другие команды, которые мы пока не будем рассматривать. Команды `try`, `catch` и `finally` используются для обработки исключений, как будет показано в главе 6. Команда `synchronized` предназначена для координации доступа к командам между несколькими программными потоками; тема синхронизации потоков рассматривается в главе 9.

Недоступные команды

И последнее замечание: компилятор Java помечает недоступные команды как ошибки компиляции. Если компилятор выясняет, что какая-либо команда не получит управления (не будет выполнена) ни при каких условиях, он помечает

¹ Переход по именованным меткам считается признаком плохого стиля.

ее как недоступную. Конечно, некоторые методы могут ни разу не вызываться в вашем коде, но компилятор обнаруживает только те из них, для которых он может «доказать» недоступность на стадии компиляции. Например, метод, в середине которого находится безусловная команда `return`, породит ошибку компиляции, как и метод с заведомо невыполнимым условием:

```
if (1 < 2) {
    // Эта ветвь выполняется всегда
    System.out.println("1 is, in fact, less than 2");
    return;
} else {
    // Недоступные команды, эта ветвь никогда не выполняется
    System.out.println("Look at that, seems we got \"math\" wrong.");
}
```

Выражения

При вычислении выражения вы получаете результат, то есть значение. Оно может относиться к числовому типу (для арифметических выражений), к ссылочному типу (при создании объектов) или к специальному типу `void`, который объявляется для методов, не возвращающих значения. В последнем случае выражение вычисляется только ради *побочных эффектов*, то есть для работы, выполняемой не с целью получения значения. Тип выражения известен во время компиляции. Значение, генерируемое во время выполнения, относится либо к этому типу, либо (для ссылочных типов) к подтипу, совместимому по присваиванию.

Выражения уже встречались вам в примерах программ и фрагментах кода. Другие примеры выражений также встречаются в разделе «Присваивание» на с. 140.

Операторы

Операторы позволяют объединять или изменять выражения различными способами. В Java поддерживаются почти все стандартные операторы из языка C. Эти операторы в Java обладают такими же приоритетами, как в C, что показано в табл. 4.3.

Таблица 4.3. Операторы Java

Приоритет	Оператор	Тип операнда	Описание
1	<code>++</code> , <code>--</code>	Арифметический	Инкремент и декремент
1	<code>+</code> , <code>-</code>	Арифметический	Унарные плюс и минус
1	<code>~</code>	Целочисленный	Поразрядное отрицание
1	<code>!</code>	Логический	Логическое отрицание

Приоритет	Оператор	Тип операнда	Описание
1	(тип)	Любой	Преобразование типа
2	*, /, %	Арифметический	Умножение, деление, остаток
3	+, -	Арифметический	Сложение и вычитание
3	+	Строковый	Конкатенация строк
4	<<	Целочисленный	Сдвиг влево
4	>>	Целочисленный	Сдвиг вправо с расширением знака
4	>>>	Целочисленный	Сдвиг вправо без расширения
5	<, <=, >, >=	Арифметический	Числовое сравнение
5	instanceof	Объект	Сравнение типов
6	==, !=	Примитивный	Проверка равенства / неравенства значений
6	==, !=	Объект	Проверка равенства / неравенства ссылок
7	&	Целочисленный	Поразрядная операция AND
7	&	Логический	Логическая операция AND
8	^	Целочисленный	Поразрядная операция XOR
8	^	Логический	Логическая операция XOR
9		Целочисленный	Поразрядная операция OR
9		Логический	Логическая операция OR
10	&&	Логический	Условная операция AND
11		Логический	Условная операция OR
12	?:	—	Условный тернарный оператор
13	=	Любой	Присваивание

Заметим, что оператор вычисления остатка % может возвращать отрицательное значение. Попробуйте поэкспериментировать с этими операторами в `jsshell`, чтобы лучше понять, как они работают. Если у вас еще нет опыта программирования, это будет особенно полезно, чтобы освоиться с операторами и их приоритетами. Выражения и операторы постоянно встречаются даже при решении самых обыденных задач в вашем коде.

```
jsshell> int x = 5
x ==> 5
```

```
jsshell> int y = 12
y ==> 12
```

```
jshell> int sumOfSquares = x * x + y * y
sumOfSquares ==> 169

jshell> int explicitOrder = ((x * x) + y) * y
explicitOrder ==> 444

jshell> sumOfSquares % 5
$7 ==> 4
```

В Java также добавлены некоторые новые операторы. Как вы уже видели, оператор `+` может использоваться со значениями `String` для выполнения конкатенации. Так как все целочисленные типы в Java имеют знак, оператор `>>` может использоваться для выполнения операции арифметического сдвига вправо с расширением знака. Оператор `>>>` интерпретирует операнд как число без знака и выполняет арифметический сдвиг вправо без расширения знака. В Java операции с отдельными битами выполняются намного реже, чем в других языках, поэтому вряд ли вы будете часто иметь дело с операторами сдвига. Если же они встретятся вам в коде, который вам попадется в интернете, запустите `jshell` и посмотрите, как они работают, или просто проанализируйте, как работает код из примера. (Это одно из наших любимых применений `jshell`!) Оператор `new` используется для создания объектов; он будет подробно рассмотрен далее.

Присваивание

Хотя инициализация переменной (то есть совмещение объявления с присваиванием) считается командой без возвращаемого значения, присваивание переменной само по себе является выражением:

```
int i, j; // Команда
i = 5; // Выражение и команда
```

Обычно присваивание используется ради его побочных эффектов, но оно также может использоваться как значение в другой части выражения:

```
j = ( i = 5 );
```

Напомним, что чрезмерная зависимость от порядка вычислений (в данном случае использование составного присваивания в сложных выражениях) затрудняет чтение кода и понимание его логики.

Значение `null`

Выражение `null` может быть присвоено любому ссылочному типу. Оно означает «ссылка отсутствует». Ссылка `null` не может использоваться для обращения к чему-либо, а любая попытка такого рода вызывает исключение `NullPointerException` во время выполнения. Вспомните, о чем говорилось в разделе «Ссылочные типы» на с. 124: `null` — это значение, присваиваемое по

умолчанию неинициализированным переменным классов и экземпляров, — обязательно инициализируйте переменные ссылочных типов перед использованием, чтобы избежать ошибок.

Обращения к переменным

Оператор «точка» (.) используется для выбора составляющих класса или экземпляра. (Они будут более подробно рассмотрены в следующих главах.) Он может использоваться, чтобы получить значение переменной экземпляра (для объекта) или статической переменной (для класса). Также он может задавать метод, который должен вызываться для объекта или класса:

```
int i = myObject.length;  
String s = myObject.name;  
myObject.someMethod();
```

Выражение ссылочного типа может использоваться в сложных вычислениях для выбора переменных или методов результата:

```
int len = myObject.name.length();  
int initialLen = myObject.name.substring(5, 10).length();
```

Здесь мы определяем длину переменной `name`, вызывая метод `length()` объекта `String`. Во втором случае выполняется промежуточный шаг, на котором мы запрашиваем подстроку строки `name`. Метод `substring` класса `String` также возвращает ссылку на объект `String`, длину которого мы запрашиваем. Подобные составные операции также называются *сцепленными* вызовами методов (см. далее). Одна из таких операций, которую мы часто использовали, — это вызов метода `println()` для переменной `out` класса `System`:

```
System.out.println( "calling println on out" );
```

Вызов методов

Методы — это функции, существующие внутри класса; они могут вызываться на уровне класса или на уровне отдельного экземпляра (в зависимости от разновидности метода). Вызов метода означает выполнение команд, составляющих его тело, с передачей всех необходимых параметров и, возможно, с возвращением значения. Вызов метода является выражением, которое возвращает значение. Тип этого значения называется *типом возвращаемого значения* метода:

```
System.out.println( "Hello, World..." );  
int myLength = myString.length();
```

Здесь методы `println()` и `length()` вызываются для разных объектов. Метод `length()` возвращает целое значение; метод `println()` возвращает `void` (зна-

чение отсутствует). Стоит заметить, что `println()` генерирует **вывод**, но не возвращает никакого **значения**. Этот метод нельзя присвоить переменной, как делалось выше с `length()`.

```
jshell> String myString = "Hi there!"
myString ==> "Hi there!"

jshell> int myLength = myString.length()
myLength ==> 9

jshell> int mistake = System.out.println("This is a mistake.")
| Error:
| incompatible types: void cannot be converted to int
| int mistake = System.out.println("This is a mistake.");
|           ^-----^
```

Методы образуют основную часть программ Java. Хотя вы можете писать некоторые простые приложения, полностью укладывающиеся в единственный в классе метод `main()`, вам вскоре станет ясно, что лучше разбивать код на части. Методы не только делают код приложения более наглядным — они открывают путь к сложным, интересным и полезным приложениям, которые попросту невозможны без них. Вспомните наши графические приложения в разделе «HelloJava» на с. 66. В них использовались методы, определенные для класса `JFrame`.

Все примеры пока были простыми, но в главе 5 вы увидите, что ситуация усложняется, когда в одном классе есть одноименные методы с разными типами параметров или когда метод переопределяется в субклассе.

Команды, выражения и алгоритмы

Итак, создадим набор команд и выражений разных типов для решения практической задачи... А проще говоря, напомним код Java для реализации алгоритма. Классическим примером считается алгоритм Евклида: нахождение наибольшего общего делителя двух чисел в простом (хотя и однообразном) процессе многократного вычитания. Для этого нам понадобится цикл `while`, условная команда `if / else` и несколько присваиваний:

```
int a = 2701;
int b = 222;
while (b != 0) {
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
System.out.println("GCD is " + a);
```

Выглядит не слишком впечатляюще, но работает — именно с такими задачами лучше всего справляются компьютерные программы. И как раз для этого вы и нужны! Мы не хотим сказать, что вы были созданы для нахождения наибольшего общего делителя чисел 2701 и 222 (кстати, он равен 37), но вся суть вашей работы — формулировка решений задач в виде алгоритмов и преобразование этих алгоритмов в исполняемый код Java. Надеемся, что еще несколько частей головоломки встали на свои места. Но даже если эти идеи вам не до конца понятны, не огорчайтесь. Процесс программирования требует серьезной практики. Попробуйте заключить приведенный выше блок кода в класс Java внутри метода `main()`. Попробуйте изменить значения `a` и `b`. В главе 8 мы займемся преобразованием строк в числа, что позволит вам находить наибольший общий делитель любых чисел, повторно запуская программу без перекомпиляции и передавая ей оба числа в параметрах метода `main()`.

Создание объектов

Объекты в Java создаются оператором `new`:

```
Object o = new Object();
```

Аргументом оператора `new` является конструктор класса. *Конструктор* представляет собой метод, имя которого всегда совпадает с именем класса. Конструктор задает все необходимые параметры для создания объекта. Значением выражения `new` является ссылка с типом создаваемого объекта. Объекты всегда имеют один или несколько конструкторов, хотя эти конструкторы не всегда доступны для вас.

Создание объектов подробно рассматривается в главе 5. А пока достаточно заметить, что создание объекта является выражением, результат которого — ссылка на этот объект. Небольшая странность заключается в том, что в другой синтаксической форме результатом выражения с оператором `new` становится возвращаемое значение метода, вызванного одновременно с созданием объекта. При необходимости вы можете создать новый объект и одновременно вызвать его метод, но не присваивать этот объект какой-либо переменной ссылочного типа:

```
int hours = new Date().getHours();
```

Вспомогательный класс `Date` здесь используется для представления текущего времени. Мы создаем новый экземпляр `Date` оператором `new` и вызываем его метод `getHours()` для получения текущего часа в виде целочисленного значения. Ссылка на объект `Date` существует достаточно долго, чтобы обеспечить вызов метода, после чего она пропадает и уничтожается уборщиком мусора в какой-то момент будущего (за информацией об уборке мусора обращайтесь к разделу «Уборка мусора» на с. 181).

Еще раз подчеркнем, что такой вызов методов по ссылкам на объекты является делом стиля. Конечно, понятнее был бы другой вариант: создать промежуточную переменную типа `Date` для хранения нового объекта, а потом вызвать его метод `getHours()`. Тем не менее подобные объединения операций встречаются достаточно часто. Когда вы освоите язык Java и будете уверенно чувствовать себя с его классами и типами, вероятно, вы возьмете на вооружение некоторые из этих приемов. А пока не беспокойтесь о том, что ваш код получается «слишком длинным». После знакомства со всем материалом этой книги вы поймете, как важно, чтобы код был «чистым» и простым для понимания.

Оператор `instanceof`

Оператор `instanceof` может использоваться для определения типа объекта во время выполнения программы. Он проверяет, относится ли объект к целевому типу или к одному из его подтипов. (Еще раз: скоро мы расскажем об иерархиях классов!) Это то же самое, что спросить, можно ли присвоить объект переменной целевого типа. Целевым типом может быть тип класса, интерфейса или массива, как будет показано далее. Оператор `instanceof` возвращает логическое значение, которое указывает, относится ли объект к этому типу:

```
boolean b;
String str = "foo";
b = ( str instanceof String ); // true, так как str относится к типу String
b = ( str instanceof Object ); // Тоже true, так как String — это Object
// b = ( str instanceof Date ); // Компилятор видит, что типы разные!
```

Оператор `instanceof` также правильно сообщает, относится ли объект к типу массива или заданного интерфейса (об этом будет рассказано далее):

```
if ( foo instanceof byte[] )
    ...
```

Важно заметить, что значение `null` не является типом какого-либо класса. Следующая проверка возвращает `false` независимо от объявленного типа переменных:

```
String s = null;
if ( s instanceof String )
    // false, так как null не является экземпляром чего-либо
```

Массивы

Массив — это особая разновидность объекта, предназначенная для хранения упорядоченного набора элементов. Тип элементов массива называется *базовым типом* массива; количество хранящихся в нем элементов называется *длиной* массива. Java поддерживает массивы всех примитивных и ссылочных типов.

Если вы занимались программированием на С или С++, то базовый синтаксис массивов выглядит очень знакомо. Мы создаем массив заданной длины и обращаемся к его элементам при помощи оператора индексирования []. Но в отличие от других языков, массивы в Java являются полноценными объектами. Массив является экземпляром специального класса `Java array` и имеет соответствующий тип в системе типов. Это означает, что для использования массива, как и любого другого объекта, необходимо сначала объявить переменную соответствующего типа, а затем создать его экземпляр оператором `new`.

Объекты массивов отличаются от других объектов Java в трех отношениях:

- Каждый раз, когда мы объявляем новый тип массива, Java неявно создает специальный тип класса массива. Чтобы пользоваться массивами, не обязательно знать этот процесс во всех тонкостях, но он поможет понять структуру массивов и их отношения с другими объектами в Java.
- Для обращения к элементам массивов используется оператор [], чтобы операции с массивами выглядели привычно. Вы также можете реализовать собственные классы, которые будут действовать как массивы, но при этом вместо специальной записи [] будут использоваться методы `get()` и `set()`.
- Java предоставляет специальную форму оператора `new`, которая позволяет сконструировать экземпляр массива заданной длины с записью [] или инициализировать его напрямую структурированным списком значений.

Типы массивов

Переменная типа массива обозначается его базовым типом, за которым следуют пустые квадратные скобки []. Также Java поддерживает объявления массива в стиле языка С, с квадратными скобками после имени массива.

Следующие объявления массива эквивалентны:

```
int [] arrayOfInts; // Рекомендуется
int arrayOfInts []; // В стиле С
```

В обоих случаях `arrayOfInts` объявляется как массив целых чисел. В этот момент размер массива еще не является проблемой, потому что мы объявляем только переменную, имеющую тип массива. Реальный экземпляр класса `array` еще не создан, память для него еще не выделена. При объявлении переменной с типом массива даже невозможно задать длину массива. Размер определяется исключительно самим объектом массива, а не ссылкой на него.

Массивы ссылочных типов создаются аналогичным способом:

```
String [] someStrings;
Button [] someButtons;
```

Создание и инициализация массива

Для создания экземпляра массива используется оператор `new`. После оператора `new` указываются базовый тип массива и его длина, определяемая целочисленным выражением в квадратных скобках:

```
arrayOfInts = new int [42];
someStrings = new String [ number + 2 ];
```

Конечно, объявление массива можно совместить с созданием объекта:

```
double [] someNumbers = new double [20];
Component [] widgets = new Component [12];
```

Индексы в массиве начинаются с нуля. Таким образом, первому элементу `someNumbers[]` соответствует индекс 0, а последнему — индекс 19. После создания все элементы массива инициализируются значением по умолчанию для своего типа. Для числовых типов это означает, что все элементы изначально равны нулю:

```
int [] grades = new int [30];
grades[0] = 99;
grades[1] = 72;
// grades[2] == 0
```

Элементы массива объектов содержат ссылки на объекты, как и отдельные переменные, на которые они ссылаются, но не содержат фактических экземпляров объектов. Следовательно, значением по умолчанию для каждого элемента будет `null`, пока вы присвоите значения экземплярам соответствующих объектов:

```
String names [] = new String [4];
names [0] = new String();
names [1] = "Walla Walla";
names [2] = someObject.toString();
// names[3] == null
```

Это важное отличие, которое может вызвать недопонимание. Во многих других языках создание массива совмещено с выделением памяти для элементов. Но в Java только что созданный массив объектов содержит только переменные ссылочного типа, каждая из которых имеет значение `null`¹. Это не означает, что для пустого массива вообще не выделяется память; она необходима для хранения

¹ Аналогом в Си или C++ является массив указателей на объекты. Указатели в этих языках являются 2-байтовыми или 4-байтовыми значениями. Создание массива указателей сводится к выделению памяти для некоторого числа объектов-указателей. Массив ссылок устроен похожим образом, но ссылки не являются объектами. Мы не можем манипулировать ссылками или их частями, кроме как путем присваивания, а объем памяти для ссылок не регламентируется в высокоуровневой спецификации языка Java.

самых этих ссылок (пустых «слотов» массива). На рис. 4.3 показан массив `names` из предыдущего примера.

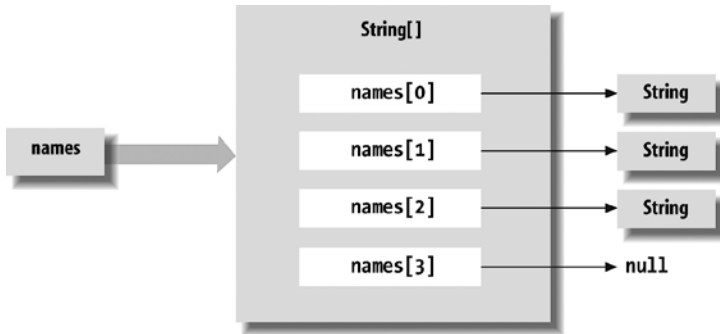


Рис. 4.3. Массив Java

Здесь `names` — это переменная типа `String[]`, то есть массив строк. Этот конкретный объект `String[]` содержит четыре переменные типа `String`. Мы присвоили объекты `String` первым трем элементам массива, а четвертый содержит значение по умолчанию `null`.

Java поддерживает конструкцию с фигурными скобками `{ }` (в стиле языка C) для создания массива и инициализации его элементов:

```
int [] primes = { 2, 3, 5, 7, 7+4 }; // Например, primes[2] = 5
```

При этом неявно создается объект массива с соответствующим типом и длиной, а значения списка выражений, разделенные запятыми, присваиваются его элементам. Обратите внимание: ключевое слово `new` и тип массива в данном случае не нужны. Тип массива был автоматически определен по присваиванию.

Синтаксис `{ }` может использоваться с массивами объектов. В этом случае при вычислении каждого выражения должен быть получен объект, который присваивается переменной базового типа массива, или значение `null`. Вот несколько примеров:

```
String [] verbs = { "run", "jump", someword.toString() };
Button [] controls = { stopButton, new Button("Forwards"),
    new Button("Backwards") };
// Все типы являются подтипами Object
Object [] objects = { stopButton, "A word", null };
```

Следующие команды эквивалентны:

```
Button [] threeButtons = new Button [3];
Button [] threeButtons = { null, null, null };
```

Использование массивов

Размер объекта массива хранится в открытой переменной `length`:

```
char [] alphabet = new char [26];
int alphaLen = alphabet.length; // alphaLen == 26
String [] musketeers = { "one", "two", "three" };
int num = musketeers.length; // num == 3
```

Здесь `length` — единственное доступное поле массива; это переменная, а не метод. (Не беспокойтесь: компилятор сообщит вам, если вы случайно добавите круглые скобки как при вызове метода; время от времени такое бывает с каждым.)

Обращения к массивам в Java практически не отличаются от обращений к массивам в других языках. Чтобы обратиться к элементу, укажите выражение с целым значением в квадратных скобках после имени массива. Следующий пример создает массив объектов `Button` с именем `keyPad`, после чего заполняет его объектами `Button`:

```
Button [] keyPad = new Button [10];
for ( int i=0; i < keyPad.length; i++ )
    keyPad[ i ] = new Button( Integer.toString( i ) );
```

Не забывайте, что для перебора значений в массиве также можно воспользоваться расширенным циклом `for`. В следующем примере он используется для вывода всех только что присвоенных значений:

```
for (Button b : keyPad)
    System.out.println(b);
```

При попытке обратиться к элементу за пределами массива генерируется исключение `ArrayIndexOutOfBoundsException`. Оно относится к типу `RuntimeException`, так что вы можете либо перехватить и обработать его самостоятельно, если ожидаете его возникновения, либо просто проигнорировать (см. главу 6). Этот пример дает некоторое представление о синтаксисе `try / catch`, который нужен для фрагментов кода с потенциальными проблемами:

```
String [] states = new String [50];
try {
    states[0] = "California";
    states[1] = "Oregon";
    ...
    states[50] = "McDonald's Land"; // Ошибка: выход за границу массива
}
catch ( ArrayIndexOutOfBoundsException err ) {
    System.out.println( "Handled error: " + err.getMessage() );
}
```

На практике часто требуется скопировать диапазон элементов из одного массива в другой. Один из способов копирования массивов основан на использовании низкоуровневого метода `arraycopy()` класса `System`:

```
System.arraycopy( source, sourceStart, destination, destStart, length );
```

В следующем примере массив `names` из предыдущего примера увеличивается вдвое:

```
String [] tmpVar = new String [ 2 * names.length ];  
System.arraycopy( names, 0, tmpVar, 0, names.length );  
names = tmpVar;
```

Новый массив, размер которого вдвое больше размера `names`, создается и присваивается временной переменной `tmpVar`. Затем метод `arraycopy()` используется для копирования элементов `names` в новый массив. Наконец, новый массив присваивается переменной `names`. Если после копирования `names` не остается ни одной ссылки на старый объект массива, он будет уничтожен в ходе следующей уборки мусора.

В другом, более простом способе копирования используются методы `java.util.Arrays.copyOf()` 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()` получает исходный массив и целевую длину. Если длина целевого массива больше длины исходного, то новый массив дополняется (нолями или `null`) до нужной длины. Метод `copyOfRange()` получает начальный индекс (включается в копируемый фрагмент) и конечный индекс (не включается), а также требуемую длину, которая тоже будет дополнена в случае необходимости.

Анонимные массивы

Часто бывает удобно создавать «одноразовые» массивы, то есть массивы, которые используются в одной точке программы (к которым программа больше нигде не обращается). Таким массивам не нужны имена, потому что вы никогда не будете

снова ссылаться на них в этом контексте. Например, вы можете создать коллекцию объектов для передачи в аргументе некоторому методу. Создать обычный именованный массив несложно, но поскольку вы никогда не будете реально работать с этим массивом (он нужен только для хранения коллекции), лучше так не делать. Java позволяет легко создавать «анонимные» (то есть неименованные) массивы.

Допустим, вы хотите вызвать метод с именем `setPets()`, которому в аргументах передается массив объектов `Animal`. Классы `Cat` и `Dog` являются subclasses `Animal`, поэтому вызов `setPets()` с помощью анонимного массива будет выглядеть так:

```
Dog pokey = new Dog ("gray");
Cat boojum = new Cat ("grey");
Cat simon = new Cat ("orange");
setPets ( new Animal [] { pokey, boojum, simon } );
```

Синтаксис похож на инициализацию массива в объявлении переменной. Мы явно определяем размер массива и заполняем его данными в фигурных скобках. Тем не менее, поскольку это не является объявлением переменной, для создания объекта массива надо явно использовать оператор `new` и тип массива.

Анонимные массивы иногда используются в качестве замены для списков аргументов переменной длины. Списки аргументов переменной длины (вероятно, знакомые программистам на языке C) предназначены для передачи методу произвольного объема данных. Представьте метод для вычисления среднего арифметического группы чисел. Вы можете поместить все числа в один массив или же сделать так, чтобы метод мог получать в аргументах одно, два, три числа и более. С появлением в Java списков аргументов переменной длины¹ полезность анонимных массивов заметно сократилась.

Многомерные массивы

Многомерные массивы поддерживаются в Java в форме массивов объектов, имеющих тип массива. Синтаксис многомерных массивов напоминает синтаксис языка C: в нем используется несколько пар квадратных скобок, по одной для каждого измерения. Этот синтаксис может использоваться для обращения к элементам в разных позициях массива. Пример многомерного массива, представляющего шахматную доску:

```
ChessPiece [][] chessBoard;
chessBoard = new ChessPiece [8][8];
chessBoard[0][0] = new ChessPiece.Rook;
chessBoard[1][0] = new ChessPiece.Pawn;
...
```

¹ Если эта идея кажется вам интересной, обратитесь к технической документации Oracle по этой теме. Попробуйте провести поиск в интернете по словам «Oracle varargs».

Здесь `chessBoard` объявляется как переменная типа `ChessPiece[][]` (то есть массив массивов `ChessPiece`). Объявление также неявно создает тип `ChessPiece[]`. Этот пример демонстрирует специальную форму оператора `new`, используемую для создания многомерных массивов. Он создает массив объектов `ChessPiece[]`, а затем поочередно создает все элементы в массиве объектов `ChessPiece`. Затем `chessBoard` индексируется для определения значений отдельных элементов `ChessPiece`. (Цвет фигур нас пока не интересует.)

Конечно, массивы могут создаваться и более чем с двумя измерениями. Приведем несколько нереальный пример:

```
Color [][][] rgbCube = new Color [256][256][256];
rgbCube[0][0][0] = Color.black;
rgbCube[255][255][0] = Color.yellow;
...
```

Мы можем указать лишь часть индексов многомерного массива (например, только индекс первого уровня), чтобы получить подмассив с меньшей размерностью. В нашем примере переменная `chessBoard` имеет тип `ChessPiece[][]`. Выражение `chessBoard[0]` обозначает первый элемент `chessBoard`, который в Java имеет тип `ChessPiece[]`. Например, шахматную доску можно заполнять по строкам:

```
ChessPiece [] homeRow = {
    new ChessPiece("Rook"), new ChessPiece("Knight"),
    new ChessPiece("Bishop"), new ChessPiece("King"),
    new ChessPiece("Queen"), new ChessPiece("Bishop"),
    new ChessPiece("Knight"), new ChessPiece("Rook")
};
chessBoard[0] = homeRow;
```

Не обязательно задавать размеры измерений в многомерном массиве одной операцией `new`. Синтаксис оператора `new` позволяет оставить размеры некоторых измерений неопределенными. Размер по крайней мере первого (самого важного) измерения массива должен быть задан обязательно, но размеры последующих, менее значимых измерений можно оставить без определения. Соответствующие значения с типами массивов можно задать впоследствии.

Таким способом можно создать шашечную доску, поля которой представлены логическими значениями (для реальной игры в шашки такой доски будет недостаточно):

```
boolean [][] checkerBoard;
checkerBoard = new boolean [8][];
```

Здесь объявляется и создается массив `checkerBoard`, но его элементы — восемь объектов `boolean[]` следующего уровня — остаются пустыми. Таким образом,

например, `checkerBoard[0]` содержит `null` до того момента, когда мы явно создадим массив и присвоим его:

```
checkerBoard[0] = new boolean [8];
checkerBoard[1] = new boolean [8];
...
checkerBoard[7] = new boolean [8];
```

Код двух предыдущих примеров эквивалентен следующему:

```
boolean [][] checkerBoard = new boolean [8][8];
```

Иногда вы можете захотеть оставить размеры массива неопределенными. Например, для того, чтобы сохранять массивы, передаваемые вам другим методом.

Поскольку длина массива не является частью его типа, представляющие шашечную доску массивы не обязаны иметь одинаковую длину; иначе говоря, многомерные массивы не обязаны быть «прямоугольными». Например, так можно представить «сломанную» (но совершенно законную с точки зрения Java) шашечную доску:

```
checkerBoard[2] = new boolean [3];
checkerBoard[3] = new boolean [10];
```

А так создается и инициализируется «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;
}
```

Типы, классы, массивы... и jshell

Java поддерживает много разных типов данных, и для каждого из них есть собственный способ представления битами или байтами в памяти. Со временем вы начнете уверенно работать с `int`, `double`, `char` и `String`. Но не спешите — именно для исследования этих фундаментальных элементов языка создавалась оболочка `jshell`. Не жалейте времени и проверяйте, правильно ли вы понимаете, что может храниться в той или иной переменной. Очень полезны эксперименты с массивами. Попробуйте разные способы объявления массивов и убедитесь в том, что вы хорошо понимаете, как обращаться к отдельным элементам в одномерных и многомерных структурах.

В `jshell` также можно экспериментировать с простыми управляющими командами, например с условными командами `if` и циклами `while`. Для ввода многострочных фрагментов кода потребуется немного терпения, но такие эксперименты и практические упражнения исключительно полезны, чтобы ваш мозг запомнил как можно больше элементов языка Java. Конечно, языки программирования проще человеческих языков, но у тех и у других есть много общего. Грамотное владение Java приобретается точно так же, как и грамотное владение английским языком (или тем языком, на котором вы читаете эту книгу). Вскоре вы начнете приблизительно понимать, что делает тот или иной код, даже если вы не понимаете всех его нюансов.

А у некоторых частей Java, в частности у массивов, таких нюансов очень много. Ранее мы упоминали, что массивы в Java являются экземплярами специальных классов. Если у массивов есть соответствующие классы, то какое место они занимают в иерархии классов и как они связаны между собой? Это хорошие вопросы, но прежде чем отвечать на них, необходимо лучше разобраться в объектно-ориентированных аспектах Java. Этой теме посвящена следующая глава. А пока поверьте на слово, что массивы занимают свое место в иерархии классов.

Объекты в Java

Пора добраться до самой сути Java и исследовать объектно-ориентированные аспекты этого языка. Термином *«объектно-ориентированное проектирование»* называется искусство разбиения приложения на объекты — самостоятельные компоненты приложения, которые работают совместно. Цель этой процедуры — разбиение задачи на множество мелких задач, более понятных и простых в решении. За много лет объектно-ориентированные архитектуры доказали свою полезность, а объектно-ориентированные языки (такие, как Java) стали хорошей основой для написания приложений — от очень маленьких до очень больших. Язык Java с самого начала проектировался как объектно-ориентированный, и все API и библиотеки Java создавались на базе проверенных паттернов объектно-ориентированного проектирования.

Объектно-ориентированная методология — это система или набор правил, упрощающих разбиение приложения на объекты. Часто это означает отображение концепций и сущностей из реального мира (иногда называемых «предметной областью задачи») на компоненты приложения. Различные методологии упрощают разложение приложения на объекты, удобные для повторного использования. Теоретически это звучит хорошо, но проблема в том, что хорошее объектно-ориентированное проектирование ближе к искусству, чем к науке. Хотя из существующих методологий проектирования можно узнать немало полезного, ни одна из них не поможет вам во всех ситуациях. Дело в том, что ничто не заменит практического опыта.

Мы не будем пытаться рекомендовать вам конкретную методологию; на эту тему и так написано множество книг¹. Вместо этого мы ограничимся советами из области здравого смысла, пока вы будете делать первые шаги.

¹ Когда вы начнете разбираться в объектно-ориентированных концепциях, вам пригодится книга Эриха Гаммы (Erich Gamma) и соавторов «Паттерны объектно-ориентированного проектирования». В ней перечислены полезные решения, доведенные до совершенства за годы практического использования. Многие из них есть в Java API.

Классы

Классы являются структурными элементами приложений Java. *Класс* может содержать методы (функции), переменные, код инициализации и, как будет показано далее, другие классы. Отдельные классы, описывающие отдельные части более сложной концепции, часто объединяются в *пакеты*, что улучшает структуру больших проектов. (Каждый класс принадлежит некоторому пакету, даже в простых примерах, которые мы рассматривали.) *Интерфейс* может описывать некоторые сходные аспекты классов, непохожих во всем остальном. Классы могут быть связаны друг с другом путем расширения, а также с интерфейсами — путем реализации. На рис. 5.1 наглядно представлены основные идеи этого очень короткого абзаца.

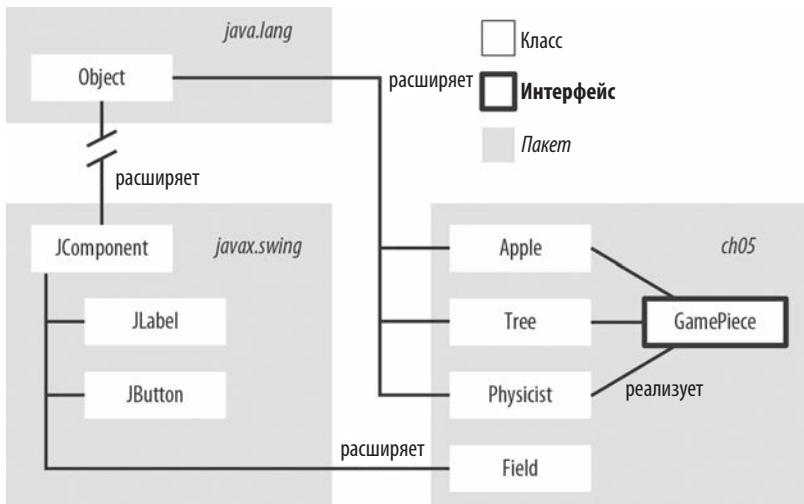


Рис. 5.1. Классы, интерфейсы и пакеты

На схеме в левом верхнем углу изображен класс `Object`. Это фундаментальный класс и на нем основаны все остальные классы Java. Класс `Object` является частью фундаментального пакета Java `java.lang`. В Java также входит пакет GUI-элементов (элементов графического интерфейса), который называется `javax.swing`. Внутри этого пакета класс `JComponent` определяет все низкоуровневые общие свойства таких графических объектов, как окна (фреймы), кнопки и холсты (canvases). Например, класс `JLabel` *расширяет* класс `JComponent`. Это означает, что `JLabel` наследует некоторые детали от `JComponent`, но добавляет к ним собственную функциональность, специфичную для надписей. Как видите, класс `JComponent` является расширением класса `Object` — возможно, не напрямую, а через промежуточные классы. Для краткости мы не стали указывать промежуточные классы и пакеты.

Программист может определять свои собственные классы и пакеты. Пакет `ch05` в правом нижнем углу — нестандартный пакет, который был написан нами. В этом пакете содержатся такие игровые классы, как `Apple` и `Field`. Также в нем содержится интерфейс `GamePiece` с некоторыми общими обязательными элементами всех игровых фигур; этот интерфейс реализуется классами `Apple`, `Tree` и `Physicist`. (В нашей игре класс `Field` представляет собой поле, на котором будут выводиться игровые фигуры, но само поле фигурой не является и поэтому **не** реализует интерфейс `GamePiece`.)

Все эти концепции подробно рассматриваются и поясняются примерами в этой главе. Важно, чтобы вы опробовали все примеры в работе и воспользовались программой `jshell` (см. раздел «Первые эксперименты с Java», с. 98), чтобы закрепить понимание новых тем.

Объявление классов и создание экземпляров

Класс служит своего рода «чертежом» для создания *экземпляров* — работающих в выполняемой программе объектов (отдельных копий), реализующих структуру класса. Класс объявляется с ключевым словом `class` и с выбранным вами именем. Например, в нашей игре **физики** бросают **яблоки** в **деревья**. Каждое существительное в этой фразе становится хорошим кандидатом для создания класса, который его будет представлять. Внутри класса мы создаем переменные со служебными данными и с другой полезной информацией, а также методы, которые описывают, что можно делать с экземплярами этого класса.

Начнем с класса, представляющего яблоки. Имена классов принято начинать с заглавной буквы — это считается обязательным! Таким образом, для этого класса хорошо подойдет имя `Apple`. Пока мы не будем заталкивать в этот класс все, что нужно знать о яблоках в нашей игре, и ограничимся несколькими элементами, которые показывают взаимодействие между классами, переменными и методами:

```
package ch05;

class Apple {
    float mass;
    float diameter = 1.0f;
    int x, y;

    boolean isTouching(Apple other) {
        ...
    }
    ...
}
```

Класс `Apple` содержит четыре переменные: `mass`, `diameter`, `x` и `y`. Он также определяет метод `isTouching()`, который получает в аргументе ссылку на другой

экземпляр `Apple` и возвращает логическое значение `boolean`. Объявления переменных и методов могут следовать в любом порядке, но при инициализации любой переменной нельзя использовать «опережающие ссылки» на другие переменные, которые появятся только в последующих строках. (В нашем маленьком фрагменте переменная `diameter` может использовать переменную `mass` для вычисления исходного значения, но `mass` не может использовать `diameter` для той же цели.) После того как класс `Apple` будет определен, можно будет создать объект `Apple` (экземпляр этого класса):

```
Apple a1;
a1 = new Apple();

// Или в одну строку:
Apple a2 = new Apple();
```

Вспомните, что объявление переменной `a1` не создает объект `Apple`; оно только создает переменную, которая ссылается на объект типа `Apple`. Затем вы должны создать объект ключевым словом `new`, как показано во второй строке приведенного выше фрагмента кода. Однако эти два шага можно объединить в одну строку, как это сделано для переменной `a2`. Конечно, однострочная запись ничего не изменит в программе: эти шаги все равно будут выполняться по отдельности. Просто объединение объявления с инициализацией иногда нагляднее.

После того как объект `Apple` будет создан, мы можем обращаться к его переменным и методам, как было показано в примерах из главы 4 и даже в графическом приложении из раздела «HelloJava» на с. 66. Например, хотя это не очень впечатляет, мы теперь можем написать другой класс `PrintAppleDetails`, который является завершенным приложением для создания экземпляра `Apple` и вывода информации о нем:

```
package ch05;

public class PrintAppleDetails {
    public static void main(String args[]) {
        Apple a1 = new Apple();
        System.out.println("Apple a1:");
        System.out.println("  mass: " + a1.mass);
        System.out.println("  diameter: " + a1.diameter);
        System.out.println("  position: (" + a1.x + ", " + a1.y + ")");
    }
}
```

Если вы скомпилируете и выполните этот пример, то в терминальном приложении или в окне терминала вашей IDE появится следующий результат:

```
Apple a1:
  mass: 0.0
  diameter: 1.0
  position: (0, 0)
```

Но почему выводится нулевое значение `mass`? Вспомните, как мы объявили переменные для класса `Apple`: там инициализируется только значение переменной `diameter`. Всем остальным переменным по умолчанию присваивается значение `0`, так как они относятся к числовым типам. (На всякий случай: логическим переменным по умолчанию присваивается значение `false`, а ссылочным — `null`.) Конечно, лучше было бы сделать более интересное «яблоко». Давайте посмотрим, как добавить к нему дополнительные данные.

Обращение к полям и методам

После того как у вас появится ссылка на объект, вы можете использовать его переменные и методы в точечной записи, как было показано в главе 4. Создадим новый класс `PrintAppleDetails2`, укажем значения `mass` и `position` для экземпляра `a1`, после чего выведем его данные:

```
package ch05;

public class PrintAppleDetails2 {
    public static void main(String args[]) {
        Apple a1 = new Apple();
        System.out.println("Apple a1:");
        System.out.println("  mass: " + a1.mass);
        System.out.println("  diameter: " + a1.diameter);
        System.out.println("  position: (" + a1.x + ", " + a1.y + ")");
        // Заполнение данных a1
        a1.mass = 10.0f;
        a1.x = 20;
        a1.y = 42;
        System.out.println("Updated a1:");
        System.out.println("  mass: " + a1.mass);
        System.out.println("  diameter: " + a1.diameter);
        System.out.println("  position: (" + a1.x + ", " + a1.y + ")");
    }
}
```

Вот что у нас получилось:

```
Apple a1:
  mass: 0.0
  diameter: 1.0
  position: (0, 0)
Updated a1:
  mass: 10.0
  diameter: 1.0
  position: (20, 42)
```

Превосходно! Теперь `a1` выглядит немного лучше. Но давайте посмотрим на код еще раз. Нам пришлось повторить три строки для вывода данных объекта.

Подобные буквальные повторения подсказывают, что здесь будет уместно определить *метод*. Методы позволяют выполнять различные операции внутри класса; они подробно рассматриваются в разделе «Методы», с. 165. Доработаем класс `Apple` и включим в него следующие команды вывода:

```
public class Apple {
    float mass;
    float diameter = 1.0f;
    int x, y;

    // ...

    public void printDetails() {
        System.out.println(" mass: " + mass);
        System.out.println(" diameter: " + diameter);
        System.out.println(" position: (" + x + ", " + y + ")");
    }

    // ...
}
```

После перемещения команд вывода в метод класс `PrintAppleDetails3` создается более компактно, чем его предшественник:

```
package ch05;

public class PrintAppleDetails3 {
    public static void main(String args[]) {
        Apple a1 = new Apple();
        System.out.println("Apple a1:");
        a1.printDetails();
        // Заполнение данных a1
        a1.mass = 10.0f;
        a1.x = 20;
        a1.y = 42;
        System.out.println("Updated a1:");
        a1.printDetails();
    }
}
```

Еще раз взгляните на метод `printDetails()`, добавленный в класс `Apple`. В коде класса можно обращаться к переменным этого класса и вызывать его методы прямо по их именам. В командах вывода на экран используются простые имена, такие как `mass` и `diameter`. Или возьмем метод `isTouching()`: координаты `x` и `y` можно использовать без специального префикса. Но чтобы обратиться к координатам какого-то другого объекта `Apple`, придется воспользоваться точечной записью. Ниже приведена реализация этого метода с математическими вычислениями (подробности — в разделе «Класс `java.lang.Math`», с. 287) и с командами `if / else`, рассмотренными в разделе «Условные команды `if / else`», с. 130:

// Файл: ch05/Apple.java

```
public boolean isTouching(Apple other) {
    double xdiff = x - other.x;
    double ydiff = y - other.y;
    double distance = Math.sqrt(xdiff * xdiff + ydiff * ydiff);
    if (distance < diameter / 2 + other.diameter / 2) {
        return true;
    } else {
        return false;
    }
}
```

Продолжим работу над игрой и создадим класс `Field`, использующий несколько объектов `Apple`. Он создает эти объекты в виде переменных экземпляра и работает с этими объектами с помощью методов `setupApples()` и `detectCollision()`, вызывая методы `Apple` и обращаясь к переменным этих объектов по ссылкам `a1` и `a2` (рис. 5.2).

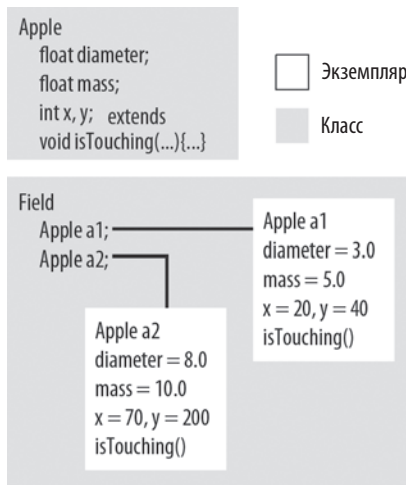


Рис. 5.2. Экземпляры класса `Apple`

```
package ch05;

public class Field {
    Apple a1 = new Apple();
    Apple a2 = new Apple();

    public void setupApples() {
        a1.diameter = 3.0f;
        a1.mass = 5.0f;
```



```

        a1.x = 20;
        a1.y = 40;
        a2.diameter = 8.0f;
        a2.mass = 10.0f;
        a2.x = 70;
        a2.y = 200;
    }

    public void detectCollisions() {
        if (a1.isTouching(a2)) {
            System.out.println("Collision detected!");
        } else {
            System.out.println("Apples are not touching.");
        }
    }
}

```

Чтобы показать, что `Field` имеет доступ к переменным и методам экземпляров класса `Apple`, давайте создадим очередную версию нашего приложения `PrintAppleDetails4`:

```

package ch05;

public class PrintAppleDetails4 {
    public static void main(String args[]) {
        Field f = new Field();
        f.setupApples();
        System.out.println("Apple a1:");
        f.a1.printDetails();
        System.out.println("Apple a2:");
        f.a2.printDetails();
        f.detectCollisions();
    }
}

```

Приложение выводит уже знакомые данные о яблоках, а затем сообщает, соприкасаются ли два яблока:

```
$ java PrintAppleDetails4
```

```

Apple a1:
  mass: 5.0
  diameter: 3.0
  position: (20, 40)
Apple a2:
  mass: 10.0
  diameter: 8.0
  position: (70, 200)
Apples are not touching.

```

Превосходно — как и предполагалось. Прежде чем читать дальше, попробуйте изменить позиции яблок, чтобы они соприкасались.

Модификаторы доступа

Несколько факторов влияют на возможность обращения к компонентам класса из другого класса. Для управления доступом можно использовать модификаторы видимости `public`, `private` и `protected`; классы также можно разместить в пакете, что влияет на их видимость. Например, модификатор `private` обозначает переменную или метод, которые должны использоваться только другими компонентами этого же класса. Допустим, в предыдущем примере при объявлении переменной `diameter` был бы указан модификатор `private`:

```
class Apple {  
    ...  
    private float diameter;  
    ...  
}
```

Теперь обратиться к переменной `diameter` из класса `Field` не удастся:

```
class Field {  
    Apple a1 = new Apple();  
    Apple a2 = new Apple();  
    ...  
    void setupApples() {  
        a1.diameter = 3.0f; // Ошибка компиляции  
        ...  
        a2.diameter = 8.0f; // Ошибка компиляции  
        ...  
    }  
    ...  
}
```

Если все же требуется обращаться извне к переменной `diameter`, то для этого в класс `Apple` надо, как это принято, включить открытые методы. Назовем их `getDiameter()` и `setDiameter()`:

```
public class Apple {  
    private float diameter = 1.0f;  
    ...  
    public void setDiameter(float newDiameter) {  
        diameter = newDiameter;  
    }  
    public float getDiameter() {  
        return diameter;  
    }  
    ...  
}
```

Создание подобных методов — хорошее правило программирования, потому что оно дает гибкие возможности для тех случаев, когда в будущем понадобится

изменить тип или смысл переменной. Далее в этой главе будут рассмотрены пакеты, модификаторы доступа и их влияние на видимость переменных и методов.

Статические поля и методы

Как было сказано выше, переменные и методы экземпляров связаны со своими экземплярами класса и доступны через эти экземпляры (то есть через конкретные объекты, такие как `a1` или `f` в приведенных примерах). С другой стороны, поля, объявленные с модификатором `static`, существуют внутри класса и совместно используются всеми экземплярами этого класса. Переменные, объявленные с модификатором `static`, называются *статическими переменными*, или *переменными класса*; методы с модификатором `static` называются *статическими методами*, или *методами класса*. Статические поля удобно использовать в качестве флагов и идентификаторов, к которым можно обратиться из любой точки. В класс `Apple` можно добавить статическую переменную, например, для хранения величины ускорения свободного падения, чтобы рассчитывать траектории брошенного яблока, когда мы начнем создавать анимацию в своей игре:

```
class Apple {  
    ...  
    static float gravAccel = 9.8f;  
    ...  
}
```

Новая переменная `float` с именем `gravAccel` объявлена с модификатором `static`. Это означает, что она существует на уровне класса, а не на уровне отдельного экземпляра, а при изменении ее значения (напрямую или через любой экземпляр `Apple`) значение изменится для всех объектов `Apple` (рис. 5.3).

К статическим полям в классе можно обращаться так же, как и к полям экземпляров. Внутри класса `Apple` мы можем использовать `gravAccel` так же, как и любую другую переменную:

```
class Apple {  
    ...  
    float getWeight () {  
        return mass * gravAccel;  
    }  
    ...  
}
```

Поскольку статические поля и методы существуют на уровне класса и не связаны ни с каким конкретным экземпляром, к ним также можно обращаться напрямую через класс. Скажем, если вам захочется покинуться яблоками на Марсе, то для чтения или присваивания переменной `gravAccel` не понадобится никакой объект

`Apple` (например, `a1` или `a2`). Вместо этого для обращения к переменной можно указать имя класса:

```
Apple.gravAccel = 3.7;
```

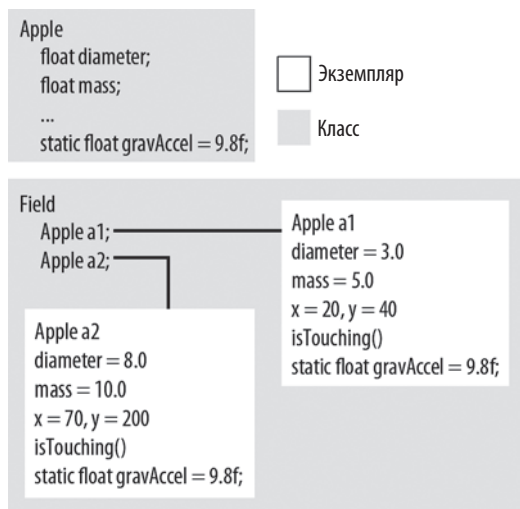


Рис. 5.3. Статические переменные совместно используются всеми экземплярами класса

Эта команда изменяет значение `gravAccel`, видимое всем экземплярам. Нам не нужно вручную задавать ускорение свободного падения для Марса в каждом экземпляре `Apple`. Статические переменные хорошо подходят для любых данных, общих для всех экземпляров класса во время выполнения. Например, можно создать методы для регистрации экземпляров, чтобы они могли взаимодействовать друг с другом или чтобы вы могли отслеживать их. Также статические переменные часто используются для определения констант. В этом случае модификатор `static` используется вместе с модификатором `final`. Таким образом, если бы вас интересовали исключительно яблоки, находящиеся под воздействием гравитационного поля Земли, класс `Apple` можно было бы изменить следующим образом:

```
class Apple {
    ...
    static final float EARTH_ACCEL = 9.8f;
    ...
}
```

Здесь мы следуем общепринятому соглашению и присваиваем константе имя, состоящее из прописных букв и символов подчеркивания (если имя состоит из нескольких слов). Значение `EARTH_ACCEL` является константой; к нему можно

обратиться из класса `Apple` или из его экземпляров, но его невозможно изменить во время выполнения.

Сочетание `static` и `final` должно использоваться только для тех данных, которые действительно должны быть неизменными. Компилятору разрешается «подставлять» эти значения в классы, которые к ним обращаются. Это означает, что при изменении переменной `static final` вам, скорее всего, придется перекомпилировать весь код, в котором эти классы используются (и это единственный случай в Java, когда возникает такая необходимость). Статические поля также хорошо подходят для значений, используемых при конструировании самого экземпляра. В нашем примере можно объявить несколько статических значений для представления различных размеров объектов `Apple`:

```
class Apple {
    ...
    static int SMALL = 0, MEDIUM = 1, LARGE = 2;
    ...
}
```

Затем эти варианты используются в методе, который задает размер для экземпляра `Apple`, или в специальном конструкторе, о котором мы расскажем позже:

```
Apple typicalApple = new Apple();
typicalApple.setSize( Apple.MEDIUM );
```

И снова внутри класса `Apple` можно обращаться к статическим полям просто по имени, а префикс `Apple.` не нужен:

```
class Apple {
    ...
    void resetEverything() {
        setSize ( MEDIUM );
    }
    ...
}
```

Методы

До настоящего момента классы в наших примерах были довольно простыми. Мы поместили в них совсем немного информации: у яблок есть масса, на поле есть несколько яблок, и т. д. Но мы также упомянули, что эти классы могут делать что-то полезное. Например, во всех классах `PrintAppleDetails` перечислялись действия, которые должны выполняться при запуске программы. Как упоминалось ранее, в Java эти действия заключаются в методы. В случае `PrintAppleDetails` это метод `main()`.

Каждый раз, когда вы выполняете некоторое действие или принимаете решение, вам нужен метод. Кроме хранения переменных (таких, как `mass` и `diameter` в классе `Apple`), мы также добавили несколько фрагментов кода, содержащих действия и логику. Методы настолько важны для работы классов, что нам пришлось создать несколько методов (вспомните метод `printDetails()` в `Apple` или метод `setupApples()` в `Field`) еще до того, как мы добрались до их формального обсуждения! Надеемся, что методы, упоминавшиеся ранее, были достаточно простыми, чтобы понять их по контексту. Но возможности методов далеко не ограничиваются выводом нескольких переменных или вычислением расстояния. Метод может содержать объявления локальных переменных и другие команды Java, выполняемые при вызове метода. Метод может возвращать значение вызывающей стороне. В методе всегда определен тип возвращаемого значения, которым может быть примитивный тип, ссылочный тип или тип `void`, обозначающий отсутствие возвращаемого значения. Метод может получать аргументы — значения, предоставляемые вызывающей стороной при вызове.

Рассмотрим простой 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 получает в аргументах два целых числа: `x` и `y`. Он возвращает значение типа `double`, для чего используется ключевое слово `return`. Наш метод получает фиксированное количество аргументов (два). Но возможны и методы со *списками аргументов переменной длины*. Такой метод может получить произвольное количество аргументов и разобраться с полученными данными во время выполнения¹.

Локальные переменные

Наш метод `fly()` объявляет локальную переменную с именем `distance`, которая используется для вычисления расстояния полета. Локальные переменные явля-

¹ В книге списки аргументов подробно не рассматриваются, но если вас заинтересует эта тема и вы захотите заняться самостоятельными исследованиями, проведите поиск в интернете по слову «varargs».

ются временными; они существуют только внутри области видимости (внутри блока) своего метода. Локальные переменные создаются в памяти при вызове метода и обычно уничтожаются при возвращении из метода. К ним нельзя обращаться за пределами самого метода. Если метод выполняется параллельно в разных потоках, то каждый поток использует собственную версию локальных переменных метода. Аргументы метода также действуют как локальные переменные в области видимости метода; единственное отличие заключается в том, что они инициализируются при передаче со стороны вызова.

Объект, созданный внутри метода и присвоенный локальной переменной, может продолжить существовать после возвращения из метода. Как будет подробно показано в разделе «Уничтожение объектов», с. 181, это зависит от того, остались ли ссылки на объект. Если объект создается, присваивается локальной переменной и больше нигде не используется, то после выхода локальной переменной из области видимости никаких ссылок на этот объект не остается, поэтому он будет уничтожен уборщиком мусора (см. раздел «Уборка мусора», с. 181). Но если мы присвоим этот объект переменной экземпляра другого объекта, или передадим его в качестве аргумента другому методу, или передадим обратно в качестве возвращаемого значения, то он может быть спасен благодаря другой переменной, содержащей ссылку на него.

Замещение

Если локальная переменная (или аргумент метода) и переменная экземпляра имеют одинаковые имена, то локальная переменная *замещает* (или скрывает) имя переменной экземпляра в области видимости метода. На первый взгляд эта ситуация выглядит странно, но она встречается довольно часто, когда переменная экземпляра имеет распространенное или очевидное имя. Например, в класс `Apple` можно добавить метод `move`, которому понадобятся новые координаты для размещения яблока. Аргументам координат будет естественно присвоить имена `x` и `y`. Однако класс уже содержит переменные экземпляра с тем же именем:

```
class Apple {
    int x, y;
    ...

    public void moveTo(int x, int y) {
        System.out.println("Moving apple to " + x + ", " + y);
        ...
    }
    ...
}
```

Если яблоко в настоящий момент находится в позиции (20, 40), а мы вызвали метод `moveTo(40, 50)`, то как вы думаете, что выведет команда `println()`? Внутри

`moveTo()` имена `x` и `y` относятся только к аргументам с указанными именами. Результат будет выглядеть так:

```
Moving apple to 40, 50
```

Если мы не можем добраться до переменных экземпляра `x` и `y`, то как же переместить яблоко? Оказывается, Java понимает суть замещения и предоставляет механизм для обхода таких ситуаций.

Ссылка `this`

Специальная ссылка `this` может использоваться каждый раз, когда вам потребуется явно обозначить текущий объект или поле текущего объекта. Часто в использовании `this` нет необходимости, потому что ссылка на текущий объект определяется неявно, например, при использовании переменных экземпляра с однозначно определяемыми именами внутри класса. Тем не менее `this` может использоваться для явных ссылок на переменные экземпляра в объекте, даже если они замещены. Следующий пример показывает, как при помощи `this` использовать имена аргументов, замещающие имена переменных экземпляра. Это довольно распространенный прием, который избавляет вас от необходимости выдумывать альтернативные имена. Реализация метода `moveTo()` с замещенными переменными могла бы выглядеть так:

```
class Apple {
    int x, y;
    ...

    public void moveTo(int x, int y) {
        System.out.println("Moving apple to " + x + ", " + y);
        this.x = x;
        if (y > diameter / 2) {
            this.y = y;
        } else {
            this.y = (int)(diameter / 2);
        }
    }
    ...
}
```

В этом примере выражение `this.x` обозначает переменную экземпляра `x` и присваивает ей значение локальной переменной `x`, которая скрывает ее имя. То же самое делается с `this.y`, но с небольшой дополнительной проверкой, которая гарантирует, что яблоко не опустится под землю. Ключевое слово `this` используется в этом примере по единственной причине: имена аргументов замещают переменные экземпляра, а мы хотим обратиться именно к переменным экземпляра. Ссылка `this` также может использоваться в ситуации, когда вам потребуется передать ссылку на «текущий» объект другому методу, как это было

сделано в графической версии приложения HelloJava в разделе «HelloJava2: продолжение», с. 79.

Статические методы

Статические методы (методы классов), как и статические переменные, принадлежат всему классу, а не отдельным экземплярам этого класса. Что это значит? Прежде всего, статический метод существует вне каких-либо конкретных экземпляров класса. Его можно вызвать по имени, через имя класса, без участия каких-либо объектов. Так как статический метод не привязан к конкретному экземпляру, он может напрямую обращаться только к другим статическим полям (статическим переменным и статическим методам) своего класса. Он не может напрямую обращаться к переменным экземпляра или вызывать методы экземпляра, потому что для этого нужно было бы сначала понять, к какому экземпляру это относится. Статические методы можно вызывать из экземпляров, используя такой же синтаксис, как и для методов экземпляров, но здесь важно то, что они также могут использоваться независимо.

Наш метод `isTouching()` использует статический метод `Math.sqrt()`, определяемый классом `java.lang.Math`. Этот класс будет подробно рассмотрен в главе 8, а пока мы подчеркнем, что `Math` — это имя класса, а не экземпляр объекта `Math`¹. Так как статические методы могут вызываться в любой ситуации, в которой доступно имя класса, они больше напоминают классические функции в стиле языка C. Статические методы особенно полезны для реализации вспомогательных методов, способных выполнять полезную работу как в привязке к конкретным экземплярам, так и без них. Например, в нашем классе `Apple` можно определить описания всех доступных размеров в виде понятных человеку строк на основе констант, созданных в разделе «Обращение к полям и методам», с. 158:

```
class Apple {
    ...

    public static String[] getAppleSizes() {
        // Возвращает имена констант
        // Порядковый номер названия должен совпадать со значением константы
        return new String[] { "SMALL", "MEDIUM", "LARGE" };
    }
    ...
}
```

¹ Класс `Math` написан так, что его экземпляры вообще не могут создаваться. Он содержит только статические методы и не имеет открытого конструктора. Попытка вызова метода `new Math()` приведет к ошибке компиляции.

Здесь определяется статический метод `getAppleSizes()`, который возвращает массив строк с названиями размеров яблок. Метод объявляется статическим, потому что список размеров остается неизменным независимо от размера любого конкретного экземпляра `Apple`. При желании метод `getAppleSizes()` можно использовать из экземпляра `Apple`, как и любой метод экземпляра. Например, метод `printDetails` (нестатический) можно изменить таким образом, чтобы он выводил название размера вместо точного значения диаметра:

```
public void printDetails() {
    System.out.println(" mass: " + mass);

    // Вывод точного значения диаметра:
    // System.out.println(" diameter: " + diameter);
    // Вывод названия (приблизительного значения диаметра):
    String niceNames[] = getAppleSizes();
    if (diameter < 5.0f) {
        System.out.println(niceNames[SMALL]);
    } else if (diameter < 10.0f) {
        System.out.println(niceNames[MEDIUM]);
    } else {
        System.out.println(niceNames[LARGE]);
    }

    System.out.println(" position: (" + x + ", " + y + ")");
}
```

Однако этот же метод может вызываться из других классов, с указанием имени класса `Apple` в точечной записи. Например, самый первый класс `PrintAppleDetails` может использовать аналогичную логику для вывода данных с использованием статического метода и статических переменных:

```
public class PrintAppleDetails {
    public static void main(String args[]) {
        String niceNames[] = Apple.getAppleSizes();
        Apple a1 = new Apple();
        System.out.println("Apple a1:");
        System.out.println(" mass: " + a1.mass);
        System.out.println(" diameter: " + a1.diameter);
        System.out.println(" position: (" + a1.x + ", " + a1.y + ")");
        if (a1.diameter < 5.0f) {
            System.out.println("This is a " + niceNames[Apple.SMALL] + " apple.");
        } else if (a1.diameter < 10.0f) {
            System.out.println("This is a " + niceNames[Apple.MEDIUM] + " apple.");
        } else {
            System.out.println("This is a " + niceNames[Apple.LARGE] + " apple.");
        }
    }
}
```

Здесь мы используем проверенный экземпляр класса `Apple` с именем `a1`, но для получения списка размеров он не нужен. Обратите внимание: список удобных названий загружается еще **до** создания `a1`. Тем не менее все успешно работает, как видно из результата:

```
Apple a1:
  mass: 0.0
  diameter: 1.0
  position: (0, 0)
This is a SMALL apple.
```

Статические методы также играют важную роль в разных паттернах программирования, в которых использование оператора `new` для класса ограничивается одним статическим методом, который называется *фабричным методом*. Создание объектов более подробно рассматривается в разделе «Конструкторы», с. 178.

Для фабричных методов не существует соглашений об именах, но на практике часто встречаются имена следующего вида:

```
Apple bigApple = Apple.createApple(Apple.LARGE);
```

Мы не будем писать фабричные методы, но вы наверняка найдете их в реальном коде, особенно при поиске ответов на таких сайтах, как `Stack Overflow`.

Инициализация локальных переменных

В отличие от переменных экземпляра, получающих значения по умолчанию, если они не были указаны явно, локальные переменные надо инициализировать перед использованием. Попытка обратиться к локальной переменной, которой не было присвоено значение, приводит к ошибке компиляции:

```
int foo;

void myMethod() {
  int bar;
  foo += 1; // Все правильно, переменная foo по умолчанию равна 0
  bar += 1; // Ошибка компиляции, переменная bar не инициализирована

  bar = 99;
  bar += 1; // Теперь правильно
}
```

Обратите внимание: это не означает, что локальная переменная должна быть инициализирована при объявлении. Просто первое обращение к ней должно быть присваиванием. Причем присваивание внутри условных команд может вызвать проблемы:

```
void myMethod {  
    int bar;  
    if ( someCondition ) {  
        bar = 42;  
        ...  
    }  
    bar += 1; // Все еще ошибка компиляции, переменная bar  
} // может быть неинициализированной
```

В этом примере переменная `bar` инициализируется только в том случае, если `someCondition` имеет значение `true`. Компилятор исключает всякий риск и помечает использование `bar` как ошибку. Ситуацию можно исправить несколькими способами. Можно заранее инициализировать переменную значением по умолчанию или переместить использование переменной в условную команду. Также можно принять меры к тому, чтобы путь выполнения не достиг неинициализированной переменной никакими способами, если это имеет смысл в нашем конкретном приложении. Например, можно проследить за тем, чтобы значение `bar` было присвоено в обеих ветвях: `if` и `else`. А еще можно вернуть управление из середины метода:

```
void myMethod {  
    int bar;  
    ...  
    if ( someCondition ) {  
        bar = 42;  
        ...  
    } else {  
        return;  
    }  
    bar += 1; // OK!  
    ...  
}
```

В этом случае нет возможности обратиться к `bar` в неинициализированном состоянии, поэтому компилятор позволяет использовать `bar` после условной команды.

Почему Java так требовательно относится к локальным переменным? Дело в том, что один из самых распространенных (и коварных) источников ошибок в других языках (вроде C и C++) — пропущенная инициализация локальных переменных, поэтому Java старается вам помочь.

Передача аргументов и ссылки

В начале главы 4 были описаны различия между примитивными типами, которые передаются по значению (посредством копирования), и объектами, которые передаются по ссылкам. Теперь, когда вы лучше умеете работать с методами в Java, рассмотрим пример:

```
void myMethod( int j, SomeKindOfObject o ) {
    ...
}
// Использование метода
int i = 0;
SomeKindOfObject obj = new SomeKindOfObject();
myMethod( i, obj );
```

Этот фрагмент кода вызывает `myMethod()` и передает ему два аргумента. Первый аргумент `i` передается по значению; при вызове метода значение `i` копируется в параметр метода (в локальную переменную с точки зрения метода) с именем `j`. Если `myMethod()` изменит значение `j`, то изменится только копия локальной переменной.

Аналогичным образом копия ссылки на `obj` помещается в ссылочную переменную `o` метода `myMethod()`. Обе ссылки относятся к одному объекту, поэтому любые изменения, вносимые по любой из ссылок, влияют на единственный существующий экземпляр объекта. Если вы измените, допустим, `o.size`, то изменение будет видимым как под именем `o.size` (внутри `myMethod()`), так и под именем `obj.size` (в вызывающем методе). Но если `myMethod()` изменит саму ссылку `o`, чтобы она указывала на другой объект, то изменение затронет только ссылку локальной переменной. Это не повлияет на переменную `obj` на стороне вызова, которая все еще относится к исходному объекту. В этом смысле передача ссылки напоминает передачу указателя в C и не похожа на передачу по ссылке в C++.

А если методу `myMethod()` также потребуется изменить смысл ссылки `obj` в вызывающем методе (то есть заставить `obj` указывать на другой объект)? Для этого проще всего упаковать `obj` в другой объект. Например, можно упаковать объект в массив как единственный элемент:

```
SomeKindOfObject [] wrapper = new SomeKindOfObject [] { obj };
```

Тогда все стороны смогут ссылаться на объект в форме `wrapper[0]` и смогут изменять ссылку. Это решение не назовешь эстетичным, но оно показывает, что вам потребуется только дополнительный уровень косвенных обращений.

Также возможен другой вариант: использование `this` для передачи ссылки вызывающему объекту. В данном случае вызывающий объект служит оберткой (`wrapper`) для ссылки. Рассмотрим следующий фрагмент, который может быть частью реализации связанного списка:

```
class Element {
    public Element nextElement;

    void addToList( List list ) {
        list.insertElement( this );
    }
}
```

```

class List {
    void insertElement( Element element ) {
        ...
        element.nextElement = getFirstElement();
        setFirstElement(element);
    }
}

```

Каждый элемент связанного списка содержит указатель на следующий элемент списка. В этом коде класс `Element` представляет один элемент; он содержит метод для включения себя в список. Класс `List` содержит метод для включения в список произвольного объекта `Element`. Метод `addToList()` вызывает `insertElement()` с аргументом `this` (который, конечно, содержит `Element`). Метод `insertElement()` может использовать полученную ссылку `this` для изменения переменной экземпляра `nextElement` объекта `Element`, а затем снова обновить начало списка. Тот же прием может использоваться в сочетании с интерфейсами для реализации обратных вызовов для произвольных методов.

Обертки для примитивных типов

Как мы отмечали в главе 4, в мире Java существует граница между типами классов (то есть объектами) и примитивными типами (числами, символами и логическими значениями). В Java это разграничение было введено по соображениям эффективности. При обработке чисел вычисления должны быть по возможности простыми; необходимость использовать объекты для примитивных типов усложнила бы оптимизацию. Но для тех ситуаций, в которых примитивные типы все-таки должны интерпретироваться как объекты, Java предоставляет стандартные классы-обертки (wrapper classes) для каждого из примитивных типов (табл. 5.1).

Таблица 5.1. Обертки для примитивных типов

Примитивный тип	Обертка
<code>void</code>	<code>java.lang.Void</code>
<code>boolean</code>	<code>java.lang.Boolean</code>
<code>char</code>	<code>java.lang.Character</code>
<code>byte</code>	<code>java.lang.Byte</code>
<code>short</code>	<code>java.lang.Short</code>
<code>int</code>	<code>java.lang.Integer</code>
<code>long</code>	<code>java.lang.Long</code>
<code>float</code>	<code>java.lang.Float</code>
<code>double</code>	<code>java.lang.Double</code>

Экземпляр класса-обертки инкапсулирует одно значение соответствующего типа. Это неизменяемый объект, который служит контейнером для хранения значения и позволяет прочитать его впоследствии. Объект-обертка строится из примитивного типа или из представления значения в формате `String`. Следующие команды эквивалентны:

```
Float pi = new Float( 3.14 );
Float pi = new Float( "3.14" );
```

Если при разборе строки происходит ошибка, то конструкторы оберток выдают исключение `NumberFormatException`.

Каждая обертка для числового типа реализует интерфейс `java.lang.Number`, который предоставляет методы для обращения к значению во всех примитивных формах. Для получения скалярных значений можно использовать методы `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 фрагмент эквивалентен преобразованию примитивного значения `double` к различным типам.

Чаще всего необходимость в обертках возникает при передаче примитивного значения такому методу, который требует передачи объекта. Например, в главе 7 будет рассмотрен API коллекций Java — набор тщательно спроектированных классов для работы с группами объектов: списками, множествами и картами. API коллекций работает с объектными типами, поэтому для сохранения в них все примитивы должны быть заключены в обертки. Скоро вы увидите, что Java может делать это автоматически. А пока сделаем это самостоятельно. Как вы увидите, `List` является расширяемой коллекцией объектов `Object`. Мы будем использовать обертки для хранения чисел в `List` (наряду с другими объектами):

```
// Простой код Java
List myNumbers = new ArrayList();
Integer thirtyThree = new Integer( 33 );
myNumbers.add( thirtyThree );
```

Здесь мы создаем обертку `Integer`, чтобы вставить число в `List` методом `add()`, который должен получать объект. Затем при извлечении элементов из `List` значение `int` можно получить следующим образом:

```
// Простой код Java
Integer theNumber = (Integer)myNumbers.get(0);
int n = theNumber.intValue(); // 33
```

Как упоминалось ранее, если вы поручите Java сделать это за вас (выполнить «автоупаковку»), то код станет более лаконичным и безопасным. Использование класса-обертки в основном скрывается от вас компилятором, но оно все еще задействовано во внутренней реализации:

```
// Код Java с использованием автоупаковки и обобщений
List<Integer> myNumbers = new ArrayList<Integer>();
myNumbers.add( 33 );
int n = myNumbers.get( 0 );
```

Работа с обобщениями будет продемонстрирована далее.

Перегрузка методов

Перегрузкой методов называется возможность определения нескольких одноименных методов в классе; при вызове метода компилятор выбирает правильный метод на основании аргументов, переданных методу. Из этого следует, что перегруженные методы должны иметь разное количество аргументов или разные типы аргументов. (В разделе «Переопределение методов», с. 194, будет рассмотрено *переопределение методов*, встречающееся при объявлении методов с совпадающими сигнатурами в subclasses.)

Перегрузка методов (также называемая *ситуационным полиморфизмом*) — это мощный и полезный инструмент. Идея заключается в создании методов, одинаково работающих с аргументами разных типов. При этом создается иллюзия того, что один метод может работать со многими типами аргументов. Метод `print()` из стандартного класса `PrintStream` показывает хороший пример перегрузки методов в действии. Как вы, вероятно, уже поняли, для вывода строкового представления чего угодно можно воспользоваться следующим выражением:

```
System.out.print( аргумент )
```

Переменная `out` представляет собой ссылку на объект (`PrintStream`), в котором определяются девять разных перегруженных версий метода `print()`. Версии получают 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 ) { ... }
    ...
}
```


Метод `print()` можно вызвать с аргументом любого из этих типов, и значение будет выведено соответствующим образом. В языке без перегрузки методов это потребовало бы более громоздкого решения — например, методов с разными именами для вывода разных типов объектов. В этом случае вы должны были бы сами разбираться, какой метод подходит для каждого из типов данных.

В предыдущем примере метод `print()` был перегружен для поддержки двух ссылочных типов: `Object` и `String`. Что будет, если вы попытаетесь вызвать `print()` с другим ссылочным типом? Например, с объектом `Date`? При отсутствии точного совпадения типа компилятор ищет допустимый вариант, совместимый *по присваиванию*. Поскольку `Date`, как и все классы, является субклассом `Object`, объект `Date` может быть присвоен переменной типа `Object`. Следовательно, этот вариант допустим и будет выбран метод `Object`.

А если возможных совпадений будет несколько? Допустим, вы хотите вывести литерал `"hi there"`. Этот литерал совместим по присваиванию либо со `String` (так как он относится к типу `String`), либо с `Object`. Здесь компилятор решает, какой вариант «лучше», и выбирает этот метод. В данном случае это метод `String`.

На интуитивном уровне это можно объяснить так: класс `String` «ближе» к литералу `"hi there"` в иерархии наследования. Такое совпадение является *более конкретным*. По другому, немного более строгому определению, первый метод *конкретнее* второго метода, если все типы аргументов первого метода совместимы по присваиванию с типами аргументов второго метода. В данном случае метод для `String` более конкретен, потому что тип `String` совместим по присваиванию с типом `Object` (обратное неверно).

Читатели, внимательно следящие за нашими объяснениями, могли заметить: мы сказали, что компилятор обрабатывает вызовы перегруженных методов. Перегрузка методов не происходит на стадии выполнения; это важное отличие. Оно означает, что выбранный метод выбирается однократно, во время компиляции. После того как перегруженный метод будет выбран, этот выбор фиксируется до возможной перекомпиляции кода, даже если класс, содержащий вызванный метод, позднее будет переработан и в него будет добавлен еще более конкретный перегруженный метод. В этом отношении перегруженные методы отличаются от *переопределенных*, которые выбираются во время выполнения и могут быть найдены даже в том случае, если они не существовали на момент компиляции вызывающего класса. На практике это отличие для вас обычно несущественно, так как вы с большой вероятностью будете перекомпилировать все необходимые классы одновременно. Переопределение методов будет рассмотрено далее в этой главе.

Создание объектов

Память для объектов в Java выделяется в системной области, которая называется *кучей* (*heap*). В отличие от других языков, вам не придется управлять

этой памятью самостоятельно. Java позаботится о выделении и освобождении памяти за вас. Java явно выделяет память в тот момент, когда объект создается оператором `new`. Что еще важнее, объекты уничтожаются уборщиком мусора, когда не остается ни одной ссылки на них.

Конструкторы

Объекты создаются оператором `new` с использованием *конструктора*. Так называется специальный метод, имя которого совпадает с именем класса, не имеющий возвращаемого типа. Конструктор вызывается при создании нового экземпляра класса, что позволяет классу подготовить этот объект к использованию. Конструкторы, как и другие методы, могут получать аргументы и перегружаться (но они не наследуются, как другие методы).

```
class Date {
    long time;

    Date() {
        time = currentTime();
    }

    Date( String date ) {
        time = parseDate( date );
    }
    ...
}
```

В этом примере класс `Date` имеет два конструктора. Первый конструктор не получает аргументов; он называется *конструктором по умолчанию*. Конструкторы по умолчанию играют особую роль: если для класса не определены никакие конструкторы, то компилятор предоставляет пустой конструктор по умолчанию. Именно конструктор по умолчанию вызывается тогда, когда вы создаете объект вызовом конструктора без аргументов. Здесь мы реализовали конструктор по умолчанию так, чтобы он задавал переменную экземпляра `time` вызовом гипотетического метода `currentTime()`, который напоминает функциональность реального класса `java.util.Date`. Второй конструктор получает аргумент `String`. Предполагается, что этот объект `String` содержит строковое представление времени, которое можно разобрать для присваивания значения переменной `time`. С конструкторами из приведенного примера объект `Date` может быть создан следующими способами:

```
Date now = new Date();
Date christmas = new Date("Dec 25, 2020");
```

В каждом случае Java выбирает на стадии компиляции подходящий конструктор на основании правил выбора перегруженных методов.

Если впоследствии все ссылки на созданный объект будут удалены, то объект будет уничтожен уборщиком мусора (об этом чуть далее):

```
christmas = null; // Кандидат для уборки мусора
```

Присваивание этой ссылке значения `null` означает, что ссылка уже не указывает на объект даты "Dec 25, 2020". Присваивание переменной `christmas` любого другого значения приведет к тому же эффекту. Если только на исходный объект даты не ссылается другая переменная, он становится недоступным и уничтожается уборщиком мусора. Мы не предлагаем присваивать ссылкам значение `null`, чтобы инициировать уничтожение объектов уборщиком мусора. Как правило, это происходит естественно: при выходе локальных переменных из области видимости. Но если на объект ссылаются переменные экземпляра в других объектах, то он будет существовать до тех пор, пока существуют эти объекты, — именно из-за этих ссылок, а статические переменные фактически существуют бесконечно.

Еще несколько замечаний: конструкторы не могут объявляться с ключевыми словами `abstract`, `synchronized` или `final` (их смысл будет объяснен далее). Тем не менее конструкторы, как и любые другие методы, могут объявляться с модификаторами видимости `public`, `private` или `protected`, управляющими их видимостью. Модификаторы видимости подробно рассматриваются в следующей главе.

Работа с перегруженными конструкторами

Конструктор может ссылаться на другой конструктор того же класса или ближайшего суперкласса при помощи специальных форм ссылок `this` и `super`. Здесь будет рассмотрен первый случай, а к конструктору суперкласса мы вернемся после рассмотрения субклассирования и наследования. Конструктор может вызвать другой перегруженный конструктор в своем классе, используя вызов автореферентного метода `this()` с соответствующими аргументами для выбора нужного конструктора. Если конструктор вызывает другой конструктор, то он должен сделать это в своей первой команде:

```
class Car {
    String model;
    int doors;

    Car( String model, int doors ) {
        this.model = model;
        this.doors = doors;
        // И прочая сложная подготовка
        ...
    }
}
```

```

    Car( String model ) {
        this( model, 4 /* двери */ );
    }
    ...
}

```

В этом примере класс `Car` имеет два конструктора. Первый, более полный, получает аргументы с моделью машины и количеством дверей. Второй конструктор получает в аргументе только модель, а затем вызывает первый конструктор со значением по умолчанию — четыре двери. У такого подхода есть важное преимущество: вся сложная подготовка может выполняться в одном конструкторе. Другие вспомогательные конструкторы просто вызывают этот конструктор, передавая ему соответствующие аргументы.

Специальный вызов `this()` должен находиться в первой команде делегирующего конструктора. Такое ограничение синтаксиса объясняется необходимостью четко определенной последовательности команд при вызове конструкторов. В конце этой последовательности Java вызовет конструктор суперкласса (если это не было сделано явно), чтобы гарантировать правильную инициализацию унаследованных полей перед продолжением работы.

Также в цепочке сразу после вызова конструктора суперкласса имеется точка, в которой вычисляются инициализаторы переменных экземпляра текущего класса. До этой точки невозможно даже обращаться к переменным экземпляра класса. Мы объясним эту ситуацию подробнее после рассмотрения темы наследования.

На данном этапе вам достаточно знать, что второй (делегированный) конструктор может вызываться только в первой команде вашего конструктора. Например, следующий код недопустим, а при попытке его скомпилировать произойдет ошибка:

```

Car( String m ) {
    int doors = determineDoors();
    this( m, doors ); // Ошибка: вызов конструктора
                    // должен быть первой командой
}

```

Простой конструктор, получающий название модели, не выполняет никаких промежуточных операций перед вызовом более явного конструктора. Он даже не может обратиться к переменной экземпляра за значением-константой:

```

class Car {
    ...
    final int defaultDoors = 4;
    ...

    Car( String m ) {

```

```
        this( m, defaultDoors ); // Ошибка: обращение
                                // к неинициализированной переменной
    }
    ...
}
```

Переменная экземпляра `defaultDoors` остается неинициализированной до более поздней точки в цепочке вызовов конструкторов, подготавливающих объект, поэтому компилятор еще не позволяет обратиться к ней. К счастью, эту конкретную проблему можно решить использованием статической переменной вместо переменной экземпляра:

```
class Car {
    ...
    static final int DEFAULT_DOORS = 4;
    ...
    Car( String m ) {
        this( m, DEFAULT_DOORS ); // Правильно!
    }
    ...
}
```

Статические поля класса инициализируются при исходной загрузке класса в виртуальную машину, поэтому обращение к ним в конструкторе безопасно.

Уничтожение объектов

Итак, теперь вы знаете, как создаются объекты. Пора поговорить об их уничтожении. Если у вас есть опыт программирования на С или С++, то, вероятно, вам пришлось провести немало времени, выявляя утечки памяти в вашем коде. Java позаботится об уничтожении объектов за вас. Вам не придется беспокоиться о привычных утечках памяти, и вы сможете сконцентрироваться на более важных задачах¹.

Уборка мусора

Для удаления объектов, которые стали ненужными программе, в Java есть механизм *уборки мусора* (*garbage collection*). Уборщик мусора — своего рода «смерть с косой». Он скрывается на заднем плане, наблюдая за объектами

¹ В Java все-таки возможно написать код, который бесконечно удерживает объекты, занимая все больше и больше памяти. Но это будет не утечка памяти, а неограниченное поглощение памяти. Обычно такие ошибки намного проще выявлять при помощи правильных инструментов и приемов.

и ожидая, пока они завершат свое существование. Уборщик мусора периодически подсчитывает ссылки на объекты и смотрит, не пришло ли их время. Когда все ссылки на объект исчезнут, а последняя возможность обратиться к объекту будет потеряна, механизм уборки мусора объявит этот объект *недоступным* и возвратит его память в пул ресурсов. Недоступным считается объект, к которому невозможно перейти по любой комбинации «живых» ссылок в работающем приложении.

При уборке мусора используется множество разных алгоритмов; архитектура виртуальной машины Java не требует применения какой-то конкретной схемы. Однако надо сказать, каким образом эта задача решается в разных реализациях Java. В ранних версиях языка использовался метод «пометки и очистки». В этой схеме Java сначала сканирует содержимое кучи в поисках непомеченных объектов. Возможность нахождения объектов в куче основана на том, что они сохраняются по особой схеме, а указатели на них содержат специальную сигнатуру битов, которая вряд ли сможет возникнуть естественным образом. У такого алгоритма нет проблемы циклических ссылок, когда объекты ссылаются друг на друга и кажутся «живыми» даже тогда, когда на самом деле они недоступны (Java решает эту проблему автоматически). Тем не менее эта схема не самая быстрая, и она создает паузы в выполнении программы. С тех пор алгоритмы уборки мусора были значительно улучшены.

Современные уборщики мусора Java работают фактически непрерывно, не вызывая никаких продолжительных задержек в выполнении приложений Java. Поскольку они являются частью исполнительской системы, они также могут совершать некоторые операции, которые невозможно выполнить статически. Например, они делят кучу на несколько областей — для объектов с разными оцениваемыми сроками жизни. Объекты с коротким сроком жизни размещаются в специальной области кучи, которая значительно сокращает время их переработки. Объекты с более долгим сроком жизни могут быть перемещены в другие, менее изменчивые части кучи. В последних реализациях уборщик мусора даже может адаптировать свое поведение, изменяя размеры частей кучи на основании фактической производительности приложения. Усовершенствование механизма уборки мусора в Java по сравнению с первыми выпусками стало весьма значительным. Это одна из причин, по которым язык Java сейчас приблизительно сравнялся по скорости со многими традиционными языками, возлагающими бремя управления памятью на программиста.

Как правило, вам не приходится беспокоиться о процессе уборки мусора. Тем не менее один метод уборки мусора может пригодиться **при отладке**. Чтобы явно порекомендовать уборщику мусора провести уборку, вызовите метод `System.gc()`. Он полностью зависит от конкретной реализации Java и в некоторых случаях может не делать ничего, но его можно вызвать, когда нужна уверенность, что Java проведет очистку перед тем, как вы выполните какую-либо операцию.

Пакеты

Даже в предыдущих простых примерах вы, возможно, заметили, что решение задач на языке Java требует создания множества классов. Классы из игры, приводившиеся выше, представляли собой яблоки, физиков, игровое поле и т. д. В более сложных приложениях или библиотеках могут использоваться сотни и даже тысячи классов. Все эти классы необходимо как-то упорядочить, и для этого в Java существует концепция *пакетов*.

Вспомните второй пример `HelloJava` из главы 2. Первые строки файла содержат информацию о местонахождении кода:

```
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 );
        ...
    }
}
```

Здесь файлу Java было присвоено имя `HelloJava`, совпадающее с именем главного класса в этом файле. Когда речь заходит о структурировании содержимого таких файлов, возникает естественная мысль: использовать иерархию папок для организации этих файлов. Именно так и поступает Java. Пакеты соответствуют именам папок, так же как классы соответствуют именам файлов. Например, посмотрите на исходный Java-код компонентов Swing, использованных в `HelloJava`, и вы найдете папку с именем `javax`, в которой находится папка `swing`, а в ней — файлы с именами `JFrame.java`, `JLabel.java` и т. д.

Импортирование классов

Одной из самых сильных сторон Java является огромное количество вспомогательных библиотек, распространяемых как на коммерческой основе, так и с открытым кодом. Потребовалось вывести содержимое PDF-файла? Для этого есть библиотека. Надо импортировать электронную таблицу? Для этого тоже есть библиотека. Включить лампочку в подвале «умного дома» с вашего веб-сервера? И для этого тоже есть своя библиотека. Возьмите любую типичную задачу для компьютеров, и вы почти всегда сможете найти библиотеку Java, которая может вам написать код для решения этой задачи.

Импортирование отдельных классов

В программировании часто действует принцип «лучше меньше, да лучше». Меньше кода — проще сопровождение. Меньше непроизводительных затрат —

выше эффективность, и т. д. (Впрочем, придерживаясь такого подхода к программированию, надо не забывать и о знаменитом изречении мудрого Эйнштейна: «Все надо делать по возможности простым, но не проще того».) Если вам нужны только один-два класса из внешнего пакета, можно импортировать только эти классы. При таком подходе ваш код станет чуть проще для понимания: другие программисты будут точно знать, какие классы вы будете использовать.

Пересмотрим приведенный выше фрагмент `HelloJava`. Ранее мы использовали массовое импортирование (подробнее об этом — в следующем разделе), но мы могли бы уточнить в коде свои намерения, импортировав только необходимые классы:

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloJava {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "Hello, Java!" );
        JLabel label = new JLabel( "Hello, Java!", JLabel.CENTER );
        ...
    }
}
```

Конечно, такой вариант импортирования занимает больше места в коде, но мы снова подчеркнем: каждый, кто будет читать или компилировать ваш код, будет точно знать существующие в нем зависимости. Во многих IDE даже поддерживается функция оптимизации импорта, которая автоматически находит такие зависимости и перечисляет их по отдельности. Если у вас появится привычка просматривать эти зависимости, вы поймете, какую пользу они приносят, когда вы разбираетесь в новом (или старом, но давно забытом) классе.

Импортирование целых пакетов

Конечно, не каждый пакет подходит для выборочного импортирования. Отличным примером служит тот же пакет `Swing`. Если вы пишете графическое десктопное приложение, то вы почти наверняка будете использовать `Swing` и его многочисленные компоненты. Для импортирования всех классов пакета используется синтаксис, уже встречавшийся вам ранее:

```
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 );
        ...
    }
}
```

Символ `*` является универсальным обозначением для импортирования классов. Эта версия команды `import` сообщает компилятору, что он должен быть готов к использованию всех классов пакета. Подобные команды импортирования

часто встречаются во многих распространенных пакетах Java, таких как AWT, Swing, Utils и пакеты ввода-вывода. Подчеркнем, что этот синтаксис подходит для любых пакетов, но в тех случаях, когда вы решите, что есть смысл определять импортируемые классы более конкретно, вы получите некоторый прирост быстродействия на стадии компиляции, а код будет легче читаться.

Отказ от импортирования

Существует еще один вариант использования внешних классов из других пакетов: не импортировать их вообще. В коде можно использовать полные имена классов. Например, наш класс `HelloJava` использует классы `JFrame` и `JLabel` из пакета `javax.swing`. При желании можно ограничиться импортированием только класса `JLabel`:

```
import javax.swing.JLabel;

public class HelloJava {
    public static void main( String[] args ) {
        javax.swing.JFrame frame = new javax.swing.JFrame( "Hello, Java!" );
        JLabel label = new JLabel( "Hello, Java!", JLabel.CENTER );
        ...
    }
}
```

На первый взгляд строка, в которой создается окно, кажется слишком длинной, но в больших классах с длинными списками импортирования одноразовое использование может сделать код более понятным. Полные имена часто указывают на то, что класс используется только в одной точке файла. (А если класс используется многократно, то его все-таки следует импортировать.) Такое использование никогда не бывает обязательным, но оно иногда встречается на практике.

Пользовательские пакеты

По мере того как вы продолжите изучать Java, писать все больше кода и решать более серьезные задачи, у вас будет накапливаться все больше классов. Для упорядочения этой коллекции можно воспользоваться пакетами. Пакеты являются ключевым словом `package`. Как упоминалось в начале этого раздела, файл с вашим классом можно разместить в структуре папок в соответствии с именем пакета. Напомним, что имена пакетов должны быть записаны в нижнем регистре, с разделением компонентов точками. Например, `javax.swing` — это имя нашего пакета графического интерфейса.

Другое соглашение, часто применяемое к именам пакетов, — так называемая система «обратных доменных имен». В отличие от пакетов, непосредственно связанных с Java, сторонние библиотеки и другой код, разработанный независимыми авторами, часто именуют на основе доменного имени компании или адреса электронной почты конкретного разработчика. Например, Mozilla Foundation

предоставляет ряд библиотек Java для сообщества разработчиков с открытым кодом. Большинство таких библиотек и вспомогательных средств оформлено в пакеты, имена которых начинаются с домена Mozilla (mozilla.org), записанного в обратном порядке: `org.mozilla`. У обратной записи имени есть удобный (и намеренный) побочный эффект: структура папок на верхнем уровне очень мала. Есть немало больших проектов, в которых используются библиотеки только из доменов верхнего уровня `com` и `org`.

Если вы пишете свои пакеты независимо от какой-либо компании или контрактного проекта, возьмите за основу свой адрес электронной почты и запишите его в обратном порядке, по аналогии с доменными именами компаний. Другой популярный вариант для кода, распространяемого в интернете, — использование домена вашего хостинг-провайдера, такого как GitHub. На GitHub публикуется огромное множество Java-проектов, создаваемых энтузиастами и любителями. Например, вы можете создать пакет с именем `com.github.myawesomeproject` (конечно, `myawesomeproject` надо заменить реальным именем проекта). Учтите, что репозитории на таких сайтах, как GitHub, часто разрешают использовать имена, недопустимые в именах пакетов. Ваш проект там может называться `my-awesome-project`, но имя пакета не может содержать дефисы ни в одной из частей. Часто такие недействительные символы просто удаляются для получения допустимого имени.

Возможно, вы уже заглянули в другие примеры из архива кода этой книги. Тогда вы наверняка заметили, что мы разместили их в соответствии с пакетами. Хотя организация классов в пакетах — неоднозначная тема, в которой не существует общепризнанных решений, мы выбрали вариант, упрощающий поиск примеров во время чтения книги. Для маленьких примеров в главах созданы пакеты вида `ch05` («глава 5»). Для примера нашей игры, проходящего через всю книгу, используется пакет `game`. Наши первые примеры можно было бы переписать, чтобы они легко укладывались в эту схему:

```
package ch02;

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 );
        ...
    }
}
```

Необходимо создать папку `ch02` и поместить в нее файл `HelloJava.java`. После этого пример можно скомпилировать и запустить в командной строке. Перейдите в начало иерархии папок и укажите полное имя файла и имя класса:

```
$ javac ch02/HelloJava.java
$ java ch02.HelloJava
```

Если вы работаете в IDE, эта среда будет управлять пакетами за вас. Просто создайте свои классы и не забудьте указать главный класс, который запускает вашу программу.

Видимость полей и методов класса

Ранее уже упоминались модификаторы доступа, которые могут использоваться при объявлении переменных и методов. Объявление чего-либо с модификатором `public` означает, что увидеть вашу переменную или вызвать ваш метод может кто угодно. Модификатор `protected` означает, что любой subclass сможет обратиться к переменной, вызвать метод или переопределить метод, чтобы предоставить альтернативную функциональность, более уместную для subclassа. Модификатор `private` означает, что переменная или метод доступны только внутри самого класса.

Пакеты влияют на доступ к защищенным (`protected`) элементам класса. Такие элементы видны и доступны не только для любого subclassа, но и для других классов того же пакета. Пакеты также учитываются, если модификатор отсутствует. Возьмем для примера текстовые компоненты из пакета `mytools.text` (рис. 5.4).

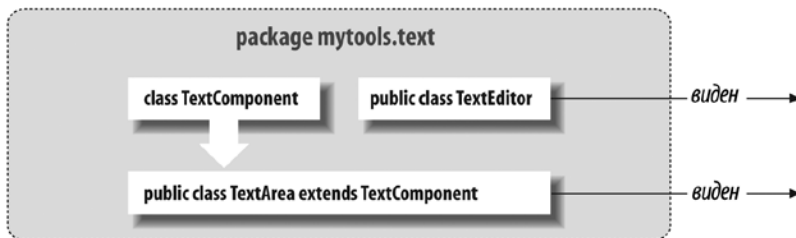


Рис. 5.4. Пакеты и видимость классов

Класс `TextComponent` не имеет модификатора. Он обладает уровнем видимости по умолчанию, или «пакетно-приватной» видимостью. Это означает, что другие классы из того же пакета могут обращаться к классу, но классы вне пакета к нему обратиться не смогут. Данная возможность может быть очень полезна для классов, относящихся ко внутренней реализации, или вспомогательных классов. Вы будете свободно пользоваться пакетно-приватными элементами, но другие программисты смогут использовать только элементы с модификаторами `public` и `protected`. На рис. 5.5 более подробно представлены возможности использования переменных и методов как subclassами, так и внешним кодом.

Обратите внимание: расширение класса `TextArea` открывает доступ как к открытым методам `getText()` и `setText()`, так и к защищенному (`protected`) методу

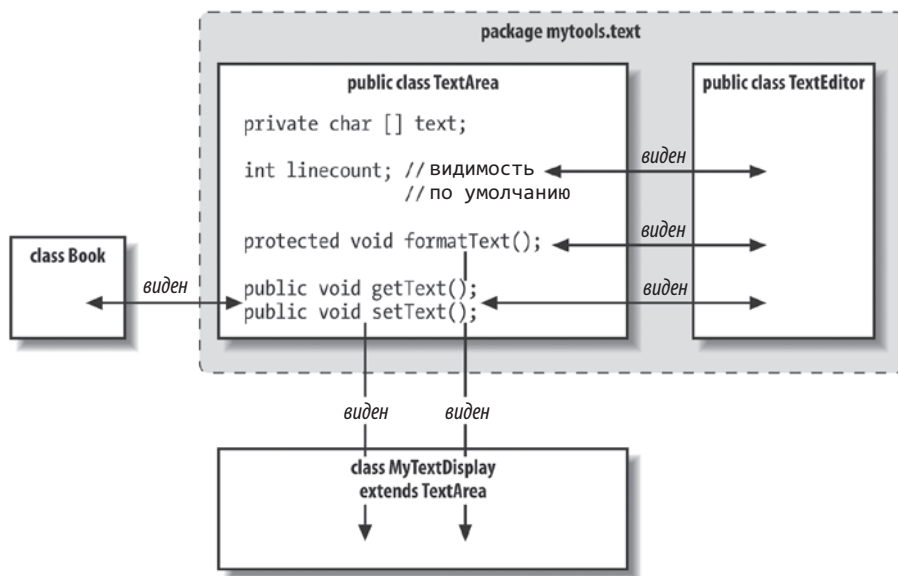


Рис. 5.5. Пакеты и видимость элементов классов

`formatText()`. При этом субкласс `MyTextDisplay` (о субклассах и ключевом слове `extends` подробно рассказано в разделе «Субклассирование и наследование», с. 190) не имеет доступа к пакетно-приватной переменной `linecount`. Однако в пакете `mytools.text`, в котором создается класс `TextEditor`, можно получить доступ к `linecount`, а также к методам с модификаторами `public` или `protected`. Внутренняя переменная для содержимого `text` остается приватной (скрытой) и недоступной для кого-либо, кроме самого класса `TextArea`.

В табл. 5.2 приведена сводка уровней видимости в Java в порядке убывания ограничений. Методы и переменные всегда видны в том классе, в котором они объявлены, поэтому в таблице этот уровень видимости не упоминается.

Таблица 5.2. Модификаторы видимости

Модификатор	Видимость за пределами класса
<code>private</code>	Нет
Без модификатора (по умолчанию)	Классы того же пакета
<code>protected</code>	Классы того же пакета и субклассы (в том же пакете или вне его)
<code>public</code>	Все классы

Компиляция с пакетами

Вы уже видели, как использовать полное имя класса при компиляции простого примера. Если вы не работаете в IDE, у вас также есть ряд других вариантов. Например, вы можете скомпилировать все классы заданного пакета таким образом:

```
$ javac ch02/*.java
$ java ch02.HelloJava
```

Учтите, что в коммерческих приложениях для предотвращения конфликтов имен часто используются более сложные имена пакетов. Одна из распространенных схем — обратная запись доменного имени вашей компании. Например, для примеров кода из книги издательства O'Reilly можно использовать полный префикс `com.oreilly.learningjava5e`. Код каждой главы преобразуется в подпакет с этим префиксом. Компиляция и запуск классов таких пакетов выполняются очень просто, хотя и не совсем компактно:

```
$ javac com/oreilly/learningjava5e/ch02/*.java
$ java com.oreilly.learningjava5e.ch02.HelloJava
```

Команда `javac` также «понимает» основные зависимости между классами. Если ваш главный класс использует несколько других классов из той же иерархии исходного кода (даже если не все они входят в один пакет), то при компиляции главного класса скомпилируются и остальные (зависимые) классы.

Впрочем, если вы не ограничиваетесь простыми программами с несколькими классами, то почти наверняка воспользуетесь IDE или программой управления сборкой (такой, как Gradle или Maven). Эти инструменты не рассматриваются в книге, но в интернете можно найти много справочников по ним. Maven отлично подходит для управления большими проектами со множеством зависимостей. За полным описанием возможностей и функциональности этой популярной программы обращайтесь к книге «Maven: the Definitive Guide» от создателя Maven Джейсона Ван Зила (Jason Van Zyl) и его команды из Sonatype (издательство O'Reilly)¹.

Нетривиальное проектирование классов

В разделе «HelloJava2: продолжение» (с. 79) два класса находились в одном файле. Это упрощало процесс компиляции, но не открывало ни одному из

¹ Программа Maven настолько изменила ситуацию с управлением зависимостями в Java и даже в других языках на базе JVM, что появились новые инструменты, базирующиеся на успехе Maven, например Gradle.

классов особого доступа к другому. Приступая к работе над более сложными задачами, вы столкнетесь со многими ситуациями, в которых нетривиальное проектирование классов, предоставляющих особый доступ, не только удобно, но и абсолютно необходимо для написания простого в сопровождении кода.

Субклассирование и наследование

Классы в языке Java образуют иерархию. Класс в Java объявляется *субклассом* (*подклассом*) другого класса при помощи ключевого слова `extends`. Субкласс *наследует* переменные и методы от своего *суперкласса* и может использовать их так, как если бы они были объявлены в самом субклассе:

```
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 переменную экземпляра `weight` и метод `eat()`. Они унаследованы от класса `Animal`.

Субкласс может быть *расширением* только одного суперкласса. В более правильной терминологии говорят, что Java поддерживает *одиночное наследование* реализации классов. Далее в этой главе будут рассматриваться интерфейсы, которые занимают место *множественного наследования*, встречающегося в других языках.

К субклассу может применяться дальнейшее субклассирование. Обычно в ходе субклассирования происходит специализация или уточнение класса путем добавления переменных и методов (но субклассирование не может привести к удалению или сокрытию переменных или методов). Пример:

```
class Cat extends Mammal {
    // Наследует weight и heartRate
```

```
boolean longHair;  
...  
  
// Наследует eat() и breathe()  
void purr() {  
    ...  
}  
}
```

Класс `Cat` относится к типу `Mammal`, который происходит от типа `Animal`. Объекты `Cat` наследуют все характеристики объектов `Mammal`, а в конечном итоге и объектов `Animal`. `Cat` также включает дополнительное поведение в виде метода `purr()` и переменной `longHair`. Этот пример отношений между классами показан на рис. 5.6.

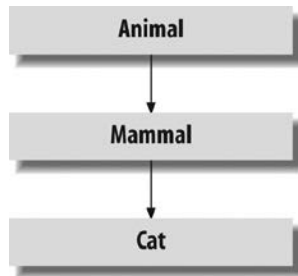


Рис. 5.6. Иерархия классов

Субкласс наследует все элементы суперкласса, не помеченные модификатором `private`. Как вы вскоре увидите, другие уровни видимости влияют на то, какие унаследованные элементы класса видны за пределами класса и его субклассов, но как минимум субкласс содержит такой же набор видимых элементов, что и его родитель. По этой причине тип субкласса может рассматриваться как подтип его родителя, а экземпляры подтипа могут использоваться везде, где могут использоваться экземпляры супертипа. Рассмотрим следующий пример:

```
Cat simon = new Cat();  
Animal creature = simon;
```

Объект `simon` класса `Cat` в этом примере может быть присвоен переменной `creature` с типом `Animal`, потому что `Cat` является подтипом `Animal`. Аналогичным образом любой метод, получающий объект `Animal`, сможет получить экземпляр `Cat` или любого типа `Mammal`. Это важный аспект полиморфизма в объектно-ориентированных языках, к числу которых относится и Java. Вы увидите, как использовать его для уточнения поведения класса и добавления к нему новой функциональности.

Замещение переменных

Как вы уже знаете, локальная переменная, имя которой совпадает с именем переменной экземпляра, *замещает* (скрывает собой) переменную экземпляра. Аналогичным образом переменная экземпляра в подклассе может заместить одноименную переменную экземпляра в родительском классе, как показано на рис. 5.7. Для полноты картины и для подготовки к изложению более сложного материала мы рассмотрим подробности сокрытия переменных, но на практике вам почти никогда не придется так поступать. Гораздо лучше структурировать код таким образом, чтобы переменные четко различались вследствие разных имен или схем выбора имен.

На рис. 5.7 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`. Выбор фактической переменной при обращении к ней в коде зависит от области видимости, в которой вы работаете, и от уточнения ссылки на нее.

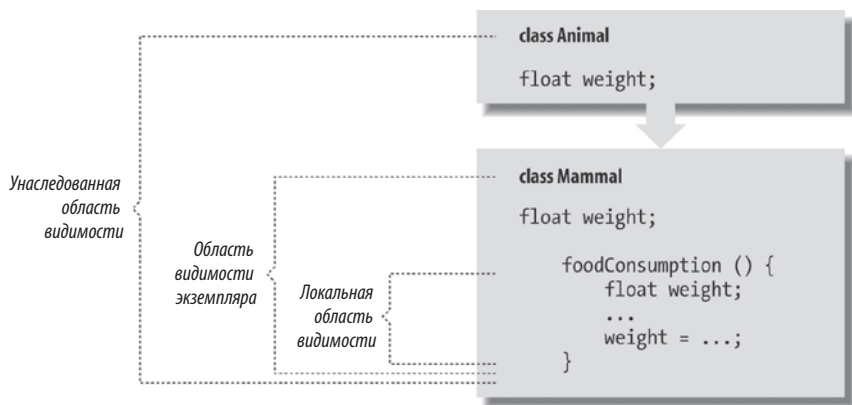


Рис. 5.7. Область видимости замещаемых переменных

В предыдущем примере все переменные относились к одному типу. В несколько более правдоподобном примере использования замещенных переменных будут изменяться их типы. Например, переменная типа `int` может замещаться переменной типа `double` в подклассе, которому требуются вещественные значения вместо целых. Это можно сделать без изменения существующего кода, потому что при замещении переменной мы ее не заменяем, а только скрываем. Обе переменные продолжают существовать: методы суперкласса видят исходную переменную, а методы подкласса — новую версию. Определение того, какую переменную видит тот или иной метод, происходит во время компиляции.

Простой пример:

```
class IntegerCalculator {
    int sum;
    ...
}

class DecimalCalculator extends IntegerCalculator {
    double sum;
    ...
}
```

В этом примере переменная экземпляра `sum` замещается таким образом, что ее тип меняется с `int` на `double`¹. Методы, определенные в классе `IntegerCalculator`, видят целочисленную переменную `sum`, а методы, определенные в классе `DecimalCalculator`, видят вещественную переменную `sum`. Тем не менее в конкретном экземпляре `DecimalCalculator` продолжают существовать обе переменные, которые могут принимать независимые значения. При этом любые методы, наследуемые классом `DecimalCalculator` от `IntegerCalculator`, видят целочисленную переменную `sum`.

Так как в `DecimalCalculator` существуют обе переменные, нам нужен способ, позволяющий ссылаться на переменную, унаследованную от `IntegerCalculator`. Для этого ссылка уточняется ключевым словом `super`:

```
int s = super.sum;
```

В классе `DecimalCalculator` ключевое слово `super`, используемое таким образом, выбирает переменную `sum`, определенную в суперклассе. Вскоре мы подробнее рассмотрим использование ключевого слова `super`.

Другое важное обстоятельство, касающееся замещения переменных, связано с тем, как они работают при обращении к объекту через родительский тип. Например, на объект `DecimalCalculator` можно сослаться как на `IntegerCalculator`, используя для этого переменную типа `IntegerCalculator`. Если вы так сделаете, а затем обратитесь к переменной `sum`, то получите целочисленную переменную вместо вещественной:

```
DecimalCalculator dc = new DecimalCalculator();
IntegerCalculator ic = dc;
int s = ic.sum; // Обращение к sum из IntegerCalculator
```

То же самое произошло бы при обращении к объекту с применением явного преобразования к типу `IntegerCalculator` или при передаче экземпляра методу, получающему родительский тип.

¹ Правильнее было бы создать абстрактный класс `Calculator` с двумя subclasses: `IntegerCalculator` и `DecimalCalculator`.

Подчеркнем, что польза от замещения переменных невелика. Значительно лучше абстрагировать использование переменных другими способами вместо того, чтобы составлять хитроумные правила видимости. Тем не менее важно понять суть происходящего, прежде чем говорить о том, как то же самое происходит с методами. Сейчас мы рассмотрим, как одни методы замещаются другими, или, в более правильной терминологии, как происходит *переопределение* методов.

Переопределение методов

Вы уже знаете, что в классе можно объявлять перегруженные методы (то есть методы с тем же именем, но с другим количеством или типом аргументов). Выбор перегруженных методов работает так, как мы рассказали, для всех методов, доступных классу, включая унаследованные. Это означает, что subclass может определять дополнительные перегруженные методы вдобавок к перегруженным методам, предоставленным суперклассом.

Кроме того, subclass может определить метод с **точно такой же** сигнатурой (с таким же именем и с такими же типами аргументов), как у метода в суперклассе. В этом случае метод subclassа *переопределяет* метод суперкласса и фактически заменяет его реализацию (рис. 5.8). Переопределение методов для изменения поведения (логики работы) объектов называется *полиморфизмом подтипов*. Именно этот механизм обычно имеют в виду, когда говорят о мощи объектно-ориентированных языков.

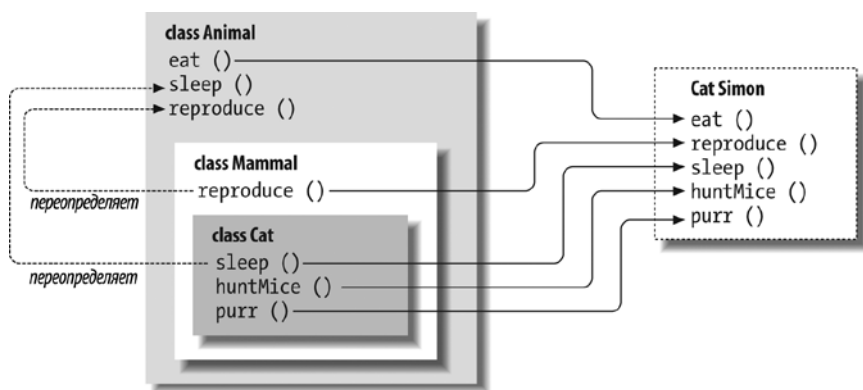


Рис. 5.8. Переопределение методов

На рис. 5.8 класс `Mammal` переопределяет метод `reproduce()` класса `Animal` — очевидно, описывающий поведение млекопитающих (`Mammal`) при рождении

новых особей¹. Для `Cat` можно также переопределить поведение сна, которое отличается от поведения обобщенной особи `Animal`, например привычку к кратковременному сну. Класс `Cat` также добавляет более уникальное поведение: мурлыканье (`purr()`) и охоту на мышей (`huntMice()`).

После всего, что вы видели до настоящего момента, может показаться, что переопределенные методы просто замещают методы суперклассов по аналогии с переменными. Но в действительности переопределенные методы обладают более широкими возможностями. Если в иерархии наследования существует несколько реализаций метода, то версия из «самого последнего» класса (то есть находящегося ниже всех в иерархии) всегда переопределяет все остальные, даже если вы ссылаетесь на объект по ссылке на тип одного из суперклассов².

Например, если имеется экземпляр `Cat`, присвоенный переменной более общего типа `Animal`, то при вызове его метода `sleep()` вы получите метод `sleep()`, реализованный в классе `Cat`, а не версию из `Animal`:

```
Cat simon = new Cat();
Animal creature = simon;
...
creature.sleep(); // Вызывается версия sleep() класса Cat
```

Иначе говоря, в отношении поведения (вызова методов) класс `Cat` ведет себя как `Cat` независимо от того, ссылаетесь ли вы на него через это имя. В остальных отношениях переменная `creature` в этом примере может вести себя как ссылка на `Animal`. Как объяснялось ранее, при обращении к замещенной переменной по ссылке `Animal` вы получите реализацию из класса `Animal`, а не из класса `Cat`. Тем не менее, поскольку поиск методов выполняется *динамически*, начиная с субклассов, будет вызван подходящий метод класса `Cat`, несмотря на то что мы работаем с ним на более общем уровне как с объектом `Animal`. Это означает, что *поведение* объектов определяется динамически. Вы можете работать со специализированными объектами так, словно они относятся к более общим типам, и при этом пользоваться преимуществами их специализированных реализаций поведения.

Интерфейсы

Интерфейсы Java расширяют концепцию абстрактных методов. Часто бывает нужно создать группу абстрактных методов, определяющих некоторое поведе-

¹ Для такого необычного яйцекладущего млекопитающего, как утконос (`platypus`), имело бы смысл снова переопределить поведение `reproduce()` в отдельном субклассе класса `Mammal`.

² Переопределенные методы в Java похожи на виртуальные методы в C++.

ние объекта без привязки к какой-либо реализации. В Java эта концепция называется интерфейсом. Интерфейс определяет набор методов, которые должны быть реализованы классом. Класс в Java может объявить, что он *реализует* интерфейс, если он реализует этот набор методов. В отличие от расширения абстрактного класса, класс, реализующий интерфейс, не обязан наследовать от какой-то конкретной части иерархии наследования или использовать конкретную реализацию.

Интерфейсы можно сравнить со скаутскими нашивками. Скаут, научившийся делать скворечники, может носить нашивку с изображением скворечника. Тем самым он сообщает окружающему миру: «Я умею строить скворечники». Аналогичным образом интерфейс представляет собой список методов, определяющих некоторое поведение объекта. Любой класс, реализующий все методы, указанные в интерфейсе, может заявить на стадии компиляции, что он реализует интерфейс. Тогда этот класс получит, словно нашивку, дополнительный тип — тип этого интерфейса.

Типы интерфейсов очень похожи на типы классов. Вы можете объявить переменную с типом интерфейса; вы можете объявить аргументы методов с типами интерфейсов; вы можете указать, что возвращаемый тип метода является типом интерфейса. В каждом случае имеется в виду, что любой объект, реализующий интерфейс (имеющий «нашивку»), подойдет на эту роль. В этом смысле интерфейсы независимы от иерархии классов. Интерфейсы выходят за границы того, к какой разновидности объектов относится некоторый объект, и взаимодействуют с ним исключительно на уровне того, что он может делать. Класс может реализовать столько интерфейсов, сколько потребуется. В этом отношении интерфейсы Java в значительной мере снимают необходимость во множественном наследовании, которое применяется в других языках (и во всех неприятных последствиях его применения).

По сути, интерфейс выглядит как чисто абстрактный класс (то есть класс, содержащий только абстрактные методы). Интерфейс определяется ключевым словом `interface` и перечислением его методов без тела — только прототипов (сигнатур):

```
interface Driveable {
    boolean startEngine();
    void stopEngine();
    float accelerate( float acc );
    boolean turn( Direction dir );
}
```

В этом примере определяется интерфейс с именем `Driveable`, состоящий из четырех методов. Методы интерфейса могут объявляться с модификатором `abstract`, хотя это и не обязательно; в данном случае мы не стали этого делать. Что еще важнее, методы интерфейса всегда считаются открытыми, и при жела-

нии их можно объявить с ключевым словом `public`. Почему открытыми? Без этого пользователь интерфейса может их попросту не увидеть, а интерфейсы обычно предназначаются для описания поведения объекта, а не его реализации.

Интерфейсы определяют функциональность, поэтому имена интерфейсов часто выбираются в соответствии с их назначением. `Driveable`, `Runnable`, `Updateable` — все эти имена хорошо подходят для интерфейсов. Любой класс, реализующий все эти методы, может объявить, что он реализует интерфейс; для этого в определении класса включается специальная секция `implements`. Пример:

```
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 о реализации `Driveable` при помощи ключевого слова `implements`.

Как показано на рис. 5.9, другой класс (например, `Lawnmower`) тоже может реализовать интерфейс `Driveable`. На диаграмме приведены реализации интерфейса `Driveable` из двух разных классов. Вероятно, что `Automobile` и `Lawnmower` происходят от некоторого класса, представляющего примитивное транспортное средство, но в данном сценарии это не обязательно.

После объявления интерфейса у вас появляется новый тип `Driveable`. Вы можете объявить переменную типа `Driveable` и присвоить ей 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();

```

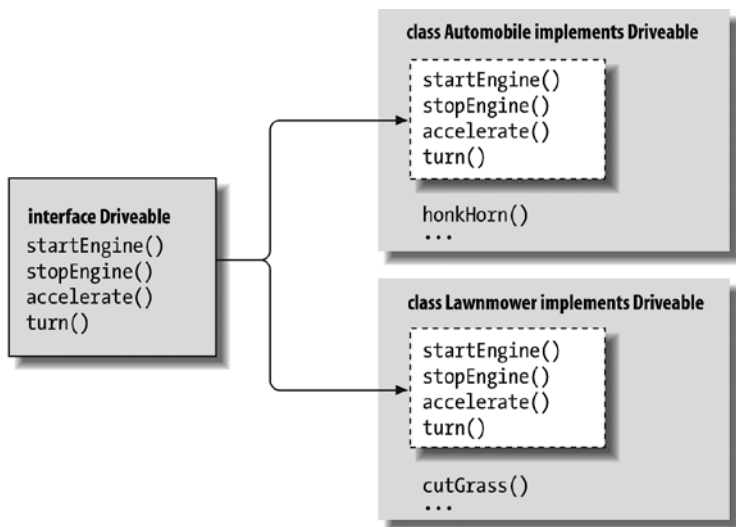


Рис. 5.9. Реализация интерфейса Driveable

Как **Automobile**, так и **Lawnmower** реализуют интерфейс **Driveable**, поэтому их можно считать взаимозаменяемыми объектами этого типа.

Внутренние классы

Все классы, встречавшиеся до настоящего момента в книге, были *высокоуровневыми*, «полноценными» классами, объявленными на уровне файлов или пакетов. Однако классы в Java могут объявляться на любом уровне видимости, в любой паре фигурных скобок (то есть почти везде, где может размещаться любая другая команда Java). Такой *внутренний класс* (*inner class*) принадлежит другому классу или методу, подобно переменной, а обращение к нему аналогичным образом может ограничиваться его областью видимости. Внутренние классы — полезное и эстетичное средство структурирования кода. Их родственники, *анонимные внутренние классы*, дают возможность еще более мощной сокращенной записи, которая выглядит так, словно вы можете динамически создавать новые виды объектов в среде Java со статической типизацией. В Java анонимные внутренние классы отчасти занимают место применяемых в других языках замыканий

(closures), реализует эффект работы с состоянием и поведением независимо от классов.

Тем не менее, когда вы начинаете разбираться во внутренних принципах их работы, становится понятно, что внутренние классы не так динамичны и элегантны, как кажется на первый взгляд. Внутренние классы представляют собой исключительно синтаксическое удобство; они не поддерживаются виртуальной машиной и вместо этого отображаются компилятором на обычные классы Java. Вам как программисту никогда не придется беспокоиться об этом; вы можете просто положить на внутренние классы как на любую другую языковую конструкцию. Тем не менее вам надо немного узнать о том, как работают внутренние классы, чтобы лучше понимать скомпилированный код и некоторые потенциальные побочные эффекты.

Внутренние классы по своей сути являются вложенными классами. Пример:

```
class Animal {
    class Brain {
        ...
    }
}
```

Здесь класс `Brain` является внутренним: он объявляется внутри области видимости класса `Animal`. И хотя смысл этого утверждения потребует некоторых пояснений, для начала скажем, что Java старается по возможности приблизить этот смысл к другим элементам (методам и переменным), находящимся на этом уровне видимости. Например, добавим метод в класс `Animal`:

```
class Animal {
    class Brain {
        ...
    }
    void performBehavior() { ... }
}
```

Как внутренний класс `Brain`, так и метод `performBehavior()` находится в области видимости `Animal`. Следовательно, в любой точке `Animal` можно обращаться к `Brain` и к `performBehavior()` напрямую, по имени. Внутри класса `Animal` можно вызвать конструктор `Brain` (`new Brain()`), чтобы получить объект `Brain`, или вызвать метод `performBehavior()` для отработки функциональности этого метода. Но ни `Brain`, ни `performBehavior()` не будут доступны за пределами класса `Animal` без дополнительного уточнения.

В теле внутреннего класса `Brain` и в теле метода `performBehavior()` мы можем обращаться ко всем остальным методам и переменным класса `Animal`. Таким образом, подобно тому как метод `performBehavior()` может работать с классом `Brain` и создавать экземпляры `Brain`, методы внутри класса `Brain` могут вызывать

метод `performBehavior()` класса `Animal`, а также работать со всеми остальными методами и переменными, объявленными в `Animal`. Класс `Brain` «видит» все методы и переменные класса `Animal` прямо в своей области видимости.

Последний факт имеет важные последствия. Из `Brain` можно вызвать метод `performBehavior()`; другими словами, из экземпляра `Brain` можно вызвать метод `performBehavior()` экземпляра `Animal`... Но из какого именно экземпляра `Animal`? Если существует несколько объектов `Animal` (допустим, несколько объектов `Cat` и `Dog`), то необходимо знать, для какого из них вызывается метод `performBehavior()`. Что это значит, когда определение класса находится «внутри» другого определения класса? Ответ: объект `Brain` всегда находится внутри одного экземпляра `Animal` — того, который был ему известен при создании. Назовем объект, содержащий какой-либо экземпляр `Brain`, его *закрывающим экземпляром*.

Объект `Brain` не может существовать вне закрывающего экземпляра объекта `Animal`. Везде, где вы видите экземпляр `Brain`, он будет привязан к экземпляру `Animal`. И хотя объект `Brain` можно сконструировать из любой точки (то есть из другого класса), `Brain` всегда требует закрывающего экземпляра `Animal`, в котором он будет «храниться». Также можно сказать, что если вы будете обращаться к `Brain` за пределами `Animal`, то такое обращение будет выглядеть как `Animal.Brain`. И как и в случае с методом `performBehavior()`, к нему можно применять модификаторы для ограничения его видимости. При этом действуют все обычные модификаторы видимости, а внутренние классы также можно объявлять статическими, как будет показано ниже.

Анонимные внутренние классы

А теперь начинается самое интересное. Как правило, чем глубже инкапсулированы и ограничены в видимости наши классы, тем больше у нас свободы для выбора их имен. Вопрос не сводится к чисто эстетическим предпочтениям. Выбор имен — важный аспект написания понятного, простого в сопровождении кода. Как правило, надо использовать настолько компактные и содержательные имена, насколько это возможно. Как следствие, желательно обойтись без изобретения имен для эфемерных объектов, которые будут использованы всего один раз.

Анонимные внутренние классы являются расширением синтаксиса операции `new`. При создании анонимного внутреннего класса его объявление объединяется с выделением памяти для экземпляра класса, что приводит к фактическому созданию «одноразового» класса и экземпляра этого класса в одной операции. После ключевого слова `new` указывается имя класса или интерфейса, за которым идет тело класса. Тело класса становится внутренним классом, который либо расширяет заданный класс, либо (в случае интерфейса) реализует интерфейс. Один экземпляр класса создается и возвращается как значение.

Например, можно вернуться к графическому приложению из раздела «HelloJava2: продолжение», с. 79. В нем создается класс `HelloComponent2`, который расширяет `JComponent` и реализует интерфейс `MouseMotionListener`. Присмотревшись к этому примеру повнимательнее, можно заметить, что `HelloComponent2` никогда не будет реагировать на события перемещения мыши, поступающие от других компонентов. Будет намного логичнее создать анонимный внутренний класс специально для перемещения надписи «Hello». В самом деле, компонент `HelloComponent2` предназначен для использования исключительно в нашем демонстрационном приложении. Можно провести рефакторинг (широко практикуемый разработчиками процесс доработки или оптимизации уже работающего кода), в результате которого класс будет выделен во внутренний класс. Теперь, когда вы больше знаете о конструкторах и наследовании, класс можно преобразовать в расширение `JFrame`, вместо того чтобы создавать окно в методе `main()`.

Ниже приведен код приложения `HelloJava3`, полученный после рефакторинга:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloJava3 extends JFrame {
    public static void main( String[] args ) {
        HelloJava3 demo = new HelloJava3();
        demo.setVisible( true );
    }

    public HelloJava3() {
        super( "HelloJava3" );
        add( new HelloComponent3("Hello, Inner Java!") );
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        setSize( 300, 300 );
    }

    class HelloComponent3 extends JComponent {
        String theMessage;
        int messageX = 125, messageY = 95; // Координаты надписи

        public HelloComponent3( String message ) {
            theMessage = message;
            addMouseMotionListener(new MouseMotionListener() {
                public void mouseDragged(MouseEvent e) {
                    messageX = e.getX();
                    messageY = e.getY();
                    repaint();
                }

                public void mouseMoved(MouseEvent e) { }
            });
        }
    }
}
```

```
        public void paintComponent( Graphics g ) {  
            g.drawString( theMessage, messageX, messageY );  
        }  
    }  
}
```

Попробуйте скомпилировать и запустить этот код. Он должен работать точно так же, как и исходное приложение `HelloJava2`. Отличается только то, как мы упорядочили классы и кто может обращаться к ним (а также к находящимся в них переменным и методам).

Систематизация кода и планирование на случай ошибок

Безусловно, классы — самая важная концепция языка Java. Они являются основой каждой исполняемой программы, портируемой библиотеки или вспомогательного модуля. Мы рассмотрели, из чего состоят классы и как они связываются друг с другом в больших проектах. Вы узнали о создании и уничтожении объектов, а в конце главы было показано, как внутренние классы (и анонимные внутренние классы) помогают писать более простой в сопровождении код. Примеры внутренних классов будут встречаться при рассмотрении более сложных тем, таких как потоки (глава 9) и Swing (глава 10).

При создании классов надо учитывать некоторые важные рекомендации:

- Старайтесь скрывать как можно большую часть реализации. Никогда не раскрывайте больше подробностей внутреннего строения объекта, чем абсолютно необходимо. Этот принцип играет ключевую роль при написании кода, простого в сопровождении и пригодного для повторного использования. Избегайте открытых переменных в своих объектах (возможно, за исключением констант). Вместо этого создавайте *методы доступа* (*accessor methods*) для чтения и записи значений (даже если они относятся к простым типам). Позднее, когда потребуется, вы сможете изменять и расширять поведение своих объектов без нарушения работоспособности других классов, которые от них зависят.
- Специализируйте объекты только там, где это необходимо, — старайтесь использовать *композицию* вместо *наследования*. Под *композицией* понимается использование объекта в его существующей форме как составляющей нового объекта. Когда вы изменяете или уточняете поведение объекта (посредством субклассирования), вы применяете *наследование*. Для повторного использования объектов надо по возможности отдавать предпочтение композиции перед наследованием, потому что при композиции объектов вы в полной мере пользуетесь возможностями существующих инструментов. Наследова-

ние приводит к нарушению инкапсуляции объектов, и применять его следует только тогда, когда оно создает реальные преимущества. Спросите себя, действительно ли вы хотите наследовать от всего класса (хотите ли вы создать новый «частный случай» этого объекта?) или же можно включить экземпляр этого класса в ваш класс и делегировать часть работы включенному объекту.

- Минимизируйте отношения между объектами и постарайтесь организовать взаимосвязанные объекты в пакеты. Классы, тесно взаимодействующие друг с другом, можно сгруппировать в пакеты Java (вспомните рис. 5.1); при этом также можно спрятать классы, не предназначенные для внешнего использования. Открывайте доступ только к классам, предназначенным для использования другими разработчиками. Чем слабее связаны ваши объекты, тем проще будет повторно использовать их в будущем.

Эти принципы желательно применять даже в небольших проектах. Примеры из папки `ch05` содержат простые версии классов и интерфейсов, которые будут нужны для создания игры с бросанием яблок. Посмотрите, как классы `Apple`, `Tree` и `Physicist` реализуют интерфейс `GamePiece` — например, метод `draw()`, присутствующий в каждом классе. Обратите внимание на то, как `Field` расширяет `JComponent` и как главный игровой класс `AppleToss` расширяет `JFrame`. На рис. 5.10 (честно говоря, пока еще не впечатляющем) показан пример

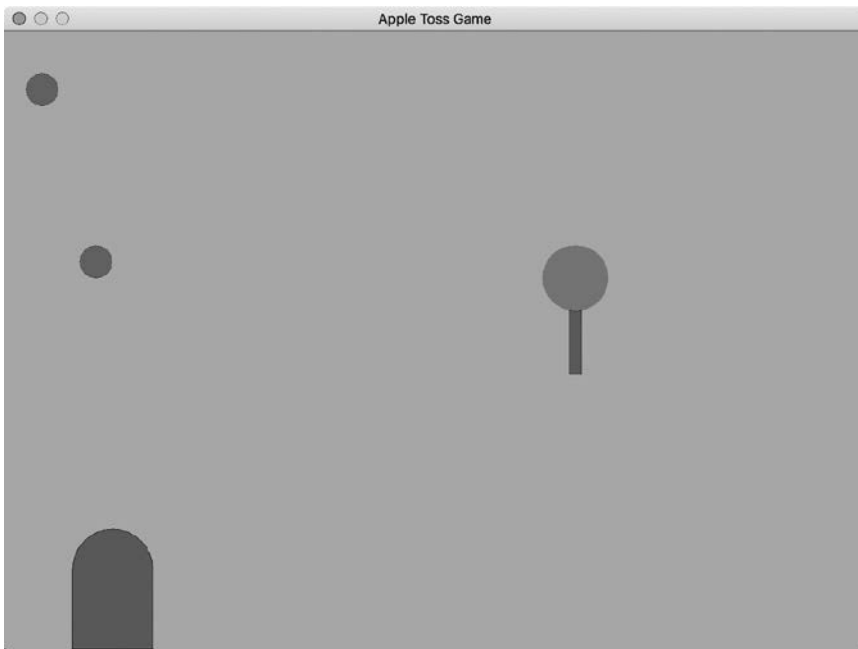


Рис. 5.10. Классы из нашей игры в действии

совместной работы этих компонентов. Чтобы опробовать его самостоятельно, скомпилируйте и запустите класс `ch05.AppleToss` так, как было описано в разделе «Пакеты», с. 183.

Просмотрите комментарии в классах. Попробуйте что-нибудь изменить. Добавьте еще одно дерево. В следующих главах мы будем развивать приложение на основе этих классов, поэтому если вы заранее разберетесь в том, как они взаимодействуют, вам будет проще понять дальнейший материал.

Как бы вы ни упорядочили элементы своих классов, классы ваших пакетов или пакеты ваших проектов, вы неизбежно столкнетесь с ошибками. Одни из них будут простыми синтаксическими ошибками, которые вы исправите в редакторе. Другие ошибки интереснее — они встречаются только во время выполнения вашей программы. Следующая глава расскажет об этих ошибках с точки зрения Java, а также поможет вам организовать их обработку.

Обработка ошибок и запись в журнал

Язык Java глубоко укоренился во встроенных системах (embedded systems). Так называется программное обеспечение мобильных компьютеров, сотовых телефонов, «умных тостеров» и многих других гаджетов, объединенных «интернетом вещей» (IoT, Internet of Things). В таких приложениях особенно важно предусмотреть надежную обработку ошибок. Многие согласятся с тем, что их телефон не должен зависать, а тосты — гореть (возможно, вместе с домом) из-за программных сбоев. Мы знаем, что нельзя полностью исключить возможность возникновения ошибок. Но выявление и методичная обработка предполагаемых ошибок на уровне приложения будут шагом в правильном направлении.

В некоторых языках всю ответственность за обработку ошибок несет программист. Сам язык никак не помогает ему распознавать ошибки и не предоставляет средства для удобной работы с ними. В языке C функция обычно сообщает об ошибке, возвращая «нереалистичное» значение (например, традиционное `-1` или `null`). А программисту надо выяснить, в чем заключается плохой результат и что он означает. Во многих случаях бывает неудобно обходить ограничения, вызванные передачей значений ошибок, сохраняя при этом нормальный путь выполнения программы¹. Есть и более сложная проблема: ошибки некоторых типов могут вполне законно возникать почти повсюду, но неразумно и невозможно добавлять соответствующие проверки в каждую точку кода.

В этой главе мы покажем, как столь актуальная «проблема всех проблем» решена в Java. Сначала мы рассмотрим понятие исключений, чтобы показать, как и по-

¹ Функции C, имеющие неочевидные имена `setjmp()` и `longjmp()`, позволяют сохранить нужную точку в процессе выполнения кода, а потом безусловно вернуться к ней, даже из очень отдаленного места. В каком-то смысле это соответствует функциональности исключений языка Java.

чему они возникают, как и где их следует обрабатывать. Также мы рассмотрим ошибки и проверочные утверждения. Ошибки — это настолько серьезные проблемы, что их, как правило, невозможно устранить во время выполнения, но можно записать в журнал (в лог-файл) для последующей отладки. Проверочные утверждения — это широко применяемый способ защиты кода от исключений и ошибок путем предварительной проверки условий, необходимых для безопасного выполнения тех или иных действий.

Исключения

В Java есть элегантный механизм *исключений*, который помогает программисту решать типичные проблемы, возникающие при написании и выполнении кода. (Обработка исключений в Java в какой-то степени похожа на обработку исключений в C++.) *Исключение* указывает на возникновение ненормального или ошибочного состояния. В этот момент программа безусловно передает управление («выдает исключение», или «выбрасывает исключение») в специально выделенную часть кода, где это состояние перехватывается и обрабатывается. Таким образом обработка ошибок выполняется независимо от нормальной последовательности выполнения программы. Вам не приходится использовать специальные возвращаемые значения во всех методах; ошибки обрабатываются отдельным механизмом. Управление может передаваться на большое расстояние от глубоко вложенных процедур, поэтому вы можете обрабатывать все ошибки в каком-то одном удобном для этого месте (или каждую ошибку — там, где она возникла). Некоторые стандартные методы Java API по-прежнему возвращают `-1` в качестве специального значения, но обычно это ограничивается теми случаями, когда вы ожидаете получить специальное значение, а ситуация не выходит за рамки допустимого¹.

В каждом методе Java должны быть объявлены все проверяемые исключения, которые он может выдавать; а компилятор следит за тем, чтобы они обрабатывались при каждом вызове этого метода. Таким образом, информация об ошибках, которые могут возникать в методе, считается настолько же важной, как типы его аргументов и возвращаемого значения. Впрочем, если вы очень захотите, то Java позволит вам игнорировать очевидные ошибки, но в таких случаях это придется делать явным образом. (Вскоре мы рассмотрим те ошибки и исключения времени выполнения, которые не обязательно объявлять или обрабатывать в методах.)

¹ Например, метод `getHeight()` класса `Image` возвращает `-1`, если высота еще неизвестна. Ошибка при этом не возникает; значение высоты может появиться в будущем. В такой ситуации незачем выдавать исключение, это снизило бы эффективность кода.

Исключения и классы ошибок

Исключения представляют собой экземпляры класса `java.lang.Exception` и его subclasses. Subclasses `Exception` могут содержать специализированную информацию (и иногда определять логику работы) для разных видов исключительных ситуаций. Но чаще всего это простые «логические» subclasses, которые нужны только для определения новых типов исключений. На рис. 6.1 показаны subclasses `Exception` из пакета `java.lang`. Эта диаграмма дает представление о том, как организованы исключения; структура классов будет подробнее рассмотрена в следующей главе. В большинстве других пакетов определяются их собственные типы исключений, которые обычно являются subclasses `Exception` или его очень важного subclasses `RuntimeException`, о котором мы вскоре расскажем.

Например, один из самых необходимых классов исключений — это `IOException` из пакета `java.io`. Класс `IOException` расширяет `Exception` и многие его subclasses для решения типичных проблем ввода-вывода (например, `FileNotFoundException`) и сетевых проблем (например, `MalformedURLException`). Сетевые исключения входят в пакет `java.net`.

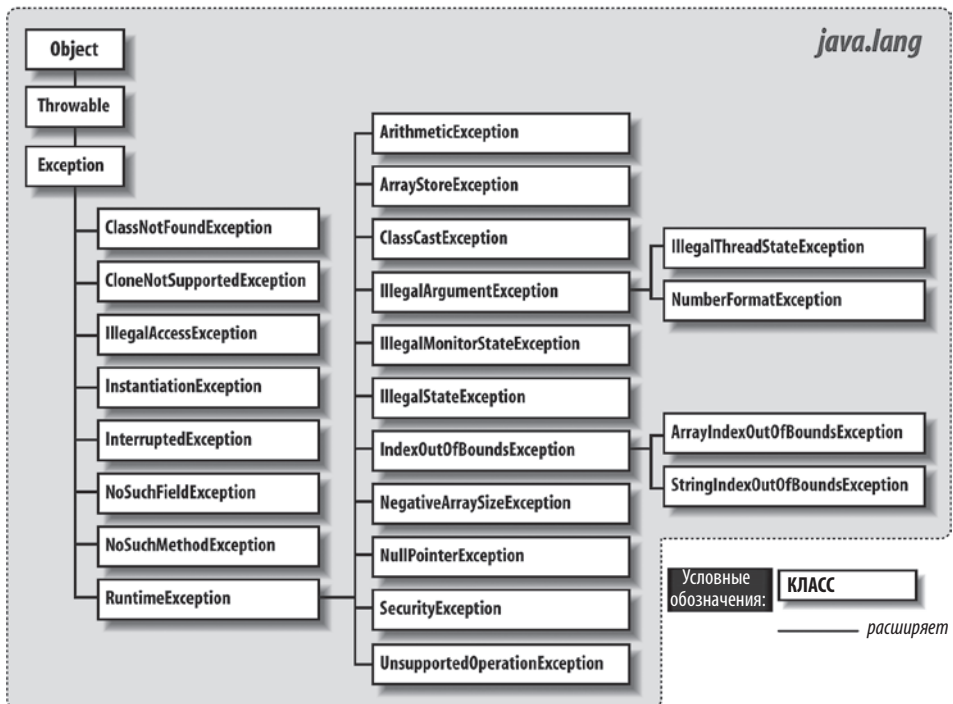


Рис. 6.1. Subclasses `java.lang.Exception`

Объект `Exception` создается в той точке, где возникло ошибочное состояние. Он может содержать любую информацию, необходимую для описания этого исключительного состояния, а также включает полную *трассировку стека* для отладки. Трассировка стека — это длинный (иногда очень длинный) список всех вызванных методов с последовательностью их вызова до точки возникновения исключения. Использование данных трассировки более подробно рассматривается в разделе «Трассировка стека», с. 213. При передаче управления объект `Exception` передается обрабатывающему блоку кода в качестве аргумента. Названия команд `throw` («выбрасывать») и `catch` («перехватывать») объясняются тем, что объект `Exception` «выбрасывается» из одной точки кода и «перехватывается» в другой точке, где выполнение продолжается.

В Java API также есть класс `java.lang.Error`, предназначенный для тех ошибок, после которых продолжение программы невозможно. Субклассы `Error` из пакета `java.lang` показаны на рис. 6.2. Среди типов `Error` заслуживает особого внимания тип `AssertionError`, используемый командой `assert` для обозначения неудачи при проверке (проверочные утверждения Java рассматриваются далее в этой главе). В некоторых других пакетах есть собственные субклассы `Error`, но вообще субклассы `Error` встречаются намного реже (и приносят меньше пользы), чем субклассы `Exception`. Когда вы пишете код, вам не надо беспокоиться об

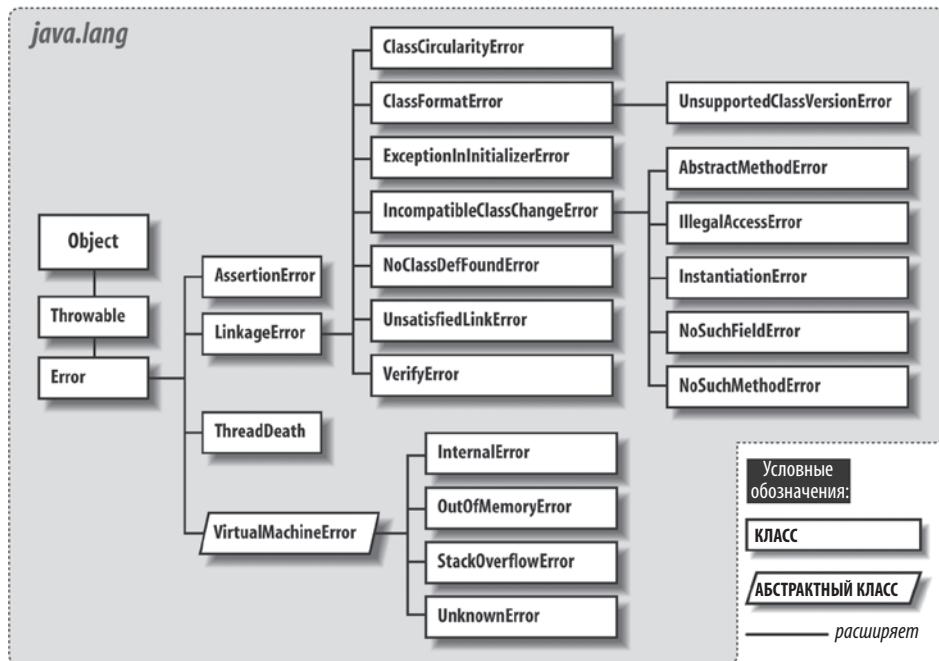


Рис. 6.2. Субклассы `java.lang.Error`

исключениях `Error` (то есть вам не придется их перехватывать); они указывают на фатальные проблемы или ошибки виртуальной машины. При возникновении такой ошибки интерпретатор Java обычно выводит сообщение и завершает работу. Даже не пробуйте перехватывать эти исключения и восстанавливать работу программы, потому что они являются признаком неустранимого сбоя, а не допустимой ситуации.

`Exception` и `Error` — это subclasses класса `Throwable`. Он является базовым классом для всех объектов, которые могут «выбрасываться» командой `throw`. Как правило, вам надо будет расширять только классы `Exception` и `Error`, а также их subclasses.

Обработка исключений

«Сторожевые» команды `try` / `catch` формируют блок кода и перехватывают возникающие в нем исключения указанных типов:

```
try {
    readFromFile("foo");
    ...
}
catch ( Exception e ) {
    // Обработка ошибки
    System.out.println( "Exception while reading file: " + e );
    ...
}
```

В этом примере все исключения, возникающие в теле секции `try`, передаются в секцию `catch`, где могут обрабатываться. Секция `catch` работает как метод, который определяет тип обрабатываемого исключения, а при вызове получает объект `Exception` в качестве аргумента. Этот объект передается в переменной `e` и «выводится на экран» вместе с сообщением.

Вспомните простую программу из главы 4, вычисляющую наибольший общий делитель по алгоритму Евклида. Сейчас мы ее доработаем, чтобы пользователь передавал два числа, `a` и `b`, в аргументах командной строки, то есть через массив `args[]` в методе `main()`. Но этот массив имеет тип `String`. Если мы немного схитрим и посмотрим на пару глав вперед, то можем использовать метод из раздела «Разбор примитивных чисел», с. 271, чтобы преобразовать эти аргументы в значения типа `int`. Тем не менее этот метод может выдать исключение, если передать ему вместо допустимого числа что-то другое. Взгляните на наш новый класс `Euclid2`:

```
public class Euclid2 {
    public static void main(String args[]) {
        int a = 2701;
        int b = 222;
```

```

// Мы будем разбирать аргументы, только если их ровно 2
if (args.length == 2) {
    try {
        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);
    } catch (NumberFormatException nfe) {
        System.err.println("Arguments were not both numbers.
            Using defaults.");
    }
} else {
    System.err.println("Wrong number of arguments (expected 2).
        Using defaults.");
}
System.out.print("The GCD of " + a + " and " + b + " is ");
while (b != 0) {
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
System.out.println(a);
}
}

```

Запустите эту программу в окне терминала (или воспользуйтесь средствами передачи аргументов командной строки в IDE, как показано на рис. 2.15), и вы сможете проверять разные пары чисел без перекомпиляции:

```
$ javac ch06/Euclid2.java
```

```
$ java ch06.Euclid2 18 6
The GCD of 18 and 6 is 6
```

```
$ java ch06.Euclid2 547832 2798
The GCD of 547832 and 2798 is 2
```

Но если при вызове будут переданы аргументы, не являющиеся числами, то вы получите исключение `NumberFormatException` и увидите сообщение об ошибке. Обратите внимание, что программа достойно реагирует на ввод некорректных значений и выводит сообщение. В этом вся суть обработки ошибок. В реальном мире ошибки неизбежны, и качество вашего кода зависит от того, как вы их обрабатываете.

```
$ java ch06.Euclid2 apples oranges
Arguments were not both numbers. Using defaults.
The GCD of 2701 and 222 is 37
```

Команда `try` может иметь несколько секций `catch`, обрабатывающих разные типы исключений (субклассы `Exception`):

```

try {
    readFromFile("foo");
    ...
}
catch ( FileNotFoundException e ) {
    // Обработка ошибки "файл не найден"
    ...
}
catch ( IOException e ) {
    // Обработка ошибки чтения
    ...
}
catch ( Exception e ) {
    // Обработка всех остальных ошибок
    ...
}

```

Секции `catch` рассматриваются по порядку, и выбирается первый подходящий вариант. При этом будет выполнена максимум одна секция `catch`; это значит, что исключения должны перечисляться в порядке от более конкретных к менее конкретным. В предыдущем примере `FileNotFoundException` является субклассом `IOException`, поэтому если бы первой секции `catch` не было, то исключение в данном случае перехватила бы вторая секция. Аналогичным образом любой субкласс `Exception` совместим с родительским типом `Exception` по присваиванию, так что третья секция `catch` перехватит все то, что пропустят первые две. Она, подобно секции `default` в команде `switch`, обрабатывает все оставшиеся возможности. Таким образом мы обеспечили полную обработку, но в общем случае старайтесь указывать типы перехватываемых исключений как можно конкретнее.

Одно из преимуществ схемы `try / catch` заключается в том, что любая команда в блоке `try` может быть уверена, что все предыдущие команды в блоке были выполнены успешно. Например, не возникнет внезапной проблемы из-за того, что программист забыл проверить возвращаемое методом значение. Если в более ранней команде произошла ошибка, то выполнение немедленно передается в секцию `catch`; последующие команды никогда не будут выполнены.

В Java 7 появилась альтернатива для использования нескольких секций `catch` — обработка нескольких типов исключений в одной секции `catch` с использованием синтаксиса `|`:

```

try {
    // Чтение из сети...
    // Запись в файл...
catch ( ZipException | SSLException e ) {
    logException( e );
}

```

Синтаксис `|` позволяет принимать оба типа исключений в общей секции `catch`. Какой же тип у переменной `e`, передаваемой нашему методу `logException()`?

(Что с ней можно сделать?) В данном случае этот тип — не `ZipException` и не `SSLException`, а `IOException` — ближайший общий предок этих двух исключений (тип ближайшего родительского класса, с которым они оба совместимы по присваиванию). Во многих подобных случаях ближайшим общим типом для нескольких исключений может быть `Exception` — родительский тип всех исключений. В отличие от простого перехвата исключений родительского типа, при перехвате нескольких исключений разных типов в общей секции `catch` мы ограничиваемся только ими, явно перечисленными, и пропускаем все остальные типы `IOException`. Вы можете использовать в секциях `catch` синтаксис `|` и одновременно упорядочивать эти секции — от самых конкретных к самым общим. Это сочетание позволит вам очень гибко их структурировать. Вы можете консолидировать всю логику обработки ошибок там, где это удобно, причем без лишних повторов кода. У этой возможности есть и другие нюансы, и мы к ней еще вернемся после обсуждения выдачи и повторной выдачи исключений.

Всплывающие исключения

А если исключение не будет перехвачено? Куда оно отправится? Если точка возникновения исключения не заключена в команду `try / catch`, то исключение «всплывает» из метода, где оно возникло, то есть передается из него в тот метод, из которого он был вызван. Если в вызывающем методе эта точка заключена в блок `try`, то управление передается соответствующей секции `catch`. В ином случае исключение продолжает всплывать вверх по стеку вызовов: от вызываемого метода к вызывающему. Таким образом исключение всплывает до тех пор, пока не будет перехвачено или пока не поднимется до самого верхнего уровня программы, что приведет к ее завершению с сообщением об ошибке времени выполнения. На самом деле ситуация немного сложнее, потому что компилятор может заставить нас решить проблему где-то в середине этого пути. В разделе «Проверяемые и непроверяемые исключения», с. 214, эта тема рассматривается более подробно.

Рассмотрим еще один пример. На рис. 6.3 показано, как метод `getContent()` вызывает метод `openConnection()` из команды `try / catch`. В свою очередь, `openConnection()` вызывает метод `sendRequest()`, который вызывает метод `write()` для отправки некоторых данных.

Как видите, во втором вызове `write()` выдается исключение `IOException`. Так как `sendRequest()` не содержит команды `try / catch` для обработки исключения, оно снова выдается в точке вызова — в методе `openConnection()`. Поскольку метод `openConnection()` тоже не перехватывает исключение, оно выдается еще раз. И в итоге оно перехватывается командой `try` в методе `getContent()` и обрабатывается его секцией `catch`. Обратите внимание: каждый метод, выдающий исключение, должен объявить его конкретный тип в секции `throws`. Мы подробнее расскажем об этом в разделе «Проверяемые и непроверяемые исключения», с. 214.

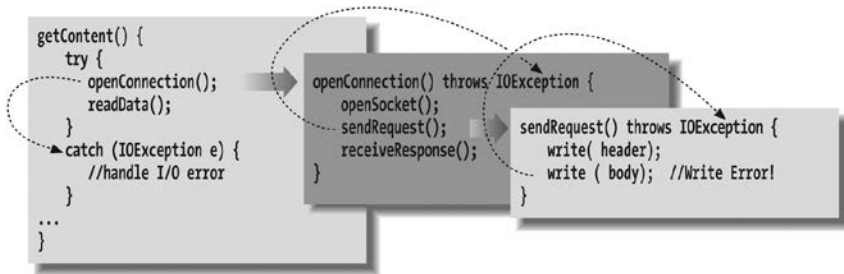


Рис. 6.3. Распространение исключений

Если вы включите высокоуровневую команду `try` в начало вашего кода, это поможет вам обработать ошибки, которые могут всплывать из фоновых потоков. Потоки рассматриваются в главе 9, а пока заметим, что перехваченные исключения могут создавать много проблем при отладке больших и сложных программ.

Трассировка стека

Исключение может пройти немалый путь, прежде чем будет перехвачено и обработано, поэтому нам нужен надежный способ находить ту точку, где оно возникло. И очень важно знать всю предысторию, то есть ту цепочку вызовов методов, которая ведет к этой точке. Для целей отладки и журналирования (логирования) каждое исключение может выводить трассировку стека, в которой указывается метод, породивший исключение, и затем все вложенные вызовы других методов. Обычно пользователь видит трассировку стека, выводимую методом `printStackTrace()`.

```

try {
    // Сложная задача со многими уровнями вложенности
} catch ( Exception e ) {
    // Вывод информации о том, где возникло исключение
    e.printStackTrace( System.err );
    ...
}

```

Например, трассировка стека для исключения может выглядеть так:

```

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)

```

Из этой трассировки видно, что метод `main()` класса `MyApplication` вызвал метод `loadFile()`. Затем метод `loadFile()` попытался создать объект `FileInputStream`,

который выдал исключение `FileNotFoundException`. Учтите, что при достижении уровня системных классов Java (таких, как `FileInputStream`) нумерация строк может пропасть. Это может случиться и при оптимизации кода в некоторых виртуальных машинах. Обычно существует возможность временного отключения оптимизации для получения точных номеров строк. Тем не менее в особенно коварных ситуациях изменение хронометража приложения может влиять на проблему, которую вы пытаетесь отслеживать, и тогда вам придется прибегнуть к другим средствам отладки.

При отладке исключений вы также можете программно получать трассировку стека с помощью метода `getStackTrace()` класса `Throwable`. (`Throwable` — базовый класс для `Exception` и `Error`.) Этот метод возвращает массив объектов `StackTraceElement`, соответствующий всему стеку вызванных методов. Подробности о них вы можете получить из `StackTraceElement`, используя методы `getFileName()`, `getClassName()`, `getMethodName()` и `getLineNumber()`. Нулевой элемент массива соответствует вершине стека, то есть той конкретной строке кода, которая породила исключение. Каждый последующий элемент массива отступает на один уровень, пока не будет достигнут исходный метод `main()`.

Проверяемые и непроверяемые исключения

Ранее мы уже упоминали, что Java заставляет нас явно выражать свои намерения относительно обработки ошибок. Но нет необходимости обрабатывать все возможные типы ошибок во всех возможных ситуациях. Поэтому исключения Java делятся на две категории: *проверяемые* и *непроверяемые*. Большинство исключений уровня приложения относится к категории проверяемых; это означает, что любой метод, выдающий исключение, — либо генерирующий его самостоятельно (см. «Выдача исключений», с. 215), либо игнорирующий произошедшее в нем исключение, — должен объявить, что он может выдавать исключение этого типа, в специальной секции `throws` в объявлении метода. Сейчас вам надо запомнить, что в методе обязательно должны быть объявлены проверяемые исключения, которые он может выдавать или передавать от других методов.

Еще раз взгляните на рис. 6.3 и обратите внимание: методы `openConnection()` и `sendRequest()` указывают, что они могут выдавать исключения типа `IOException`. Если метод может выдавать исключения нескольких типов, то их можно перечислить, разделяя запятыми:

```
void readFile( String s ) throws IOException, InterruptedException {  
    ...  
}
```

Секция `throws` сообщает компилятору, что метод является возможным источником проверяемого исключения и к его обработке должны быть готовы все,

кто вызывает этот метод. Вызывающая сторона должна либо использовать блок `try / catch` для обработки, либо в свою очередь объявить, что она сама может выдать это исключение.

Исключения, являющиеся subclasses классов `java.lang.RuntimeException` и `java.lang.Error`, относятся к категории непроверяемых. На рис. 6.1 показаны subclasses `RuntimeException`. (Subclasses `Error` обычно резервируются для серьезных проблем с загрузкой классов или с исполнительной системой.) Если вы игнорируете возможность этих исключений, это не приведет к ошибке компиляции; методы также не обязаны объявлять, что они могут выдавать эти исключения. Во всех остальных отношениях непроверяемые исключения ведут себя так же, как и другие исключения. Вы можете перехватывать их, если хотите, но обычно в этом нет необходимости.

Проверяемые исключения предназначены для проблем уровня приложения, таких как отсутствие файлов и недоступность веб-серверов. Вы как хороший программист (и добропорядочный гражданин) должны проектировать свой код так, чтобы он корректно восстанавливался в подобных ситуациях. Непроверяемые исключения предназначены для проблем системного уровня, таких как нехватка памяти или выход за границы массива. Хотя эти ситуации могут свидетельствовать об ошибках программирования на уровне приложения, они могут возникать где угодно, и обычно восстановление после них невозможно. К счастью, поскольку эти исключения являются непроверяемыми, вам не придется заключать в конструкцию `try / catch` каждую операцию с индексом массива (или объявлять все вызывающие методы как потенциальные источники таких исключений).

Подведем итог. Проверяемые исключения — это проблемы, которые нормальное приложение должно всегда обрабатывать корректно. Непроверяемые исключения (исключения времени выполнения и ошибки) — это проблемы, на восстановление после которых ваше приложение обычно не может рассчитывать. Исключения типа `Error` означают фатальные ошибки, то есть в нормальном приложении вам уже бессмысленно их обрабатывать, стараясь восстановить работоспособность программы.

Выдача исключений

Вы можете выдавать собственные исключения — экземпляры `Exception`, экземпляры одного из существующих subclasses или ваших собственных специализированных классов исключений. Все, что для этого нужно, — создать экземпляр `Exception` и выдать его командой `throw`:

```
throw new IOException();
```

Выполнение кода останавливается, и управление передается ближайшей замыкающей команде `try / catch`, которая может обработать этот тип исключения.

(Вероятно, хранить ссылку на созданный здесь объект нет смысла.) Альтернативный конструктор позволяет указать строку с сообщением об ошибке:

```
throw new IOException("Sunspots!");
```

Для получения этой строки можно воспользоваться методом `getMessage()` объекта `Exception`. Впрочем, часто можно просто вывести сам объект исключения (или результат вызова `toString()`), чтобы получить сообщение и трассировку стека.

По соглашению все типы `Exception` имеют конструктор `String` такого вида. Приведенное выше сообщение `String` особой пользы не приносит. Обычно используется более конкретный субкласс `Exception`, предоставляющий подробную информацию или по крайней мере более конкретное строковое описание. Другой пример:

```
public void checkRead( String s ) {
    if ( new File(s).isAbsolute() || (s.indexOf("..") != -1) )
        throw new SecurityException(
            "Access to file : "+ s +" denied.");
}
```

В этом коде частично реализуется метод для проверки недопустимого пути. Если такой путь будет найден, то выдается исключение `SecurityException` с информацией о нарушении.

Конечно, в специализированные субклассы `Exception` можно включать любую другую информацию, которая может оказаться полезной. Но чаще всего вам будет достаточно просто иметь новый тип исключения, чтобы правильно передать управление. Например, при разработке парсера (анализатора текста) вы можете создать собственный тип исключения для обозначения конкретной разновидности сбоя:

```
class ParseException extends Exception {
    private int lineNumber;

    ParseException() {
        super();
        this.lineNumber = -1;
    }

    ParseException( String desc, int lineNumber ) {
        super( desc );
        this.lineNumber = lineNumber;
    }

    public int getLineNumber() {
        return lineNumber;
    }
}
```


О классах и конструкторах мы подробно рассказывали в разделе «Конструкторы» на с. 82. Тело нашего класса (субкласса `Exception`) в этом примере позволяет создавать исключение `ParseException` — так же, как мы создавали исключения ранее (обычные или с небольшой дополнительной информацией). Теперь, когда у нас есть новый тип исключения, мы можем защититься от сбоя таким образом:

```
// Где-то в коде
...
try {
    parseStream( input );
} catch ( ParseException pe ) {
    // Некорректный ввод...
    // Мы даже можем назвать строку, где возникла проблема!
} catch ( IOException ioe ) {
    // Низкоуровневая проблема с каналом связи
}
```

Как видите, даже без детальной информации (например, без номера строки, в которой возникла проблема со входными данными) наше собственное исключение помогло отличить ошибку разбора данных от произвольной ошибки ввода-вывода в том же фрагменте кода.

Сцепление и повторная выдача исключений

Иногда требуется выполнить некоторые действия в зависимости от исключения, а затем выдать вместо него новое исключение. Такое решение часто встречается при проектировании фреймворков, в которых низкоуровневые детализированные исключения обрабатываются и заменяются исключениями более высокого уровня, с которыми проще работать. Представьте, что вам надо перехватить исключение `IOException` в коммуникационном пакете, затем выполнить завершающие действия (например, освободить некоторые ресурсы), а в итоге выдать собственное высокоуровневое исключение (например, `LostServerConnection`).

Очевидное решение — просто перехватить исключение, а затем выдать новое. Но тогда вы потеряете важную информацию, в том числе трассировку стека исходного исключения. Чтобы этого не произошло, вы можете воспользоваться механизмом *сцепления исключений*. Это означает, что вы должны соединить исходное исключение с новым исключением, которое выдаете. В Java есть явная поддержка сцепления исключений. При создании экземпляра базового класса `Exception` в аргументе может передаваться исключение или же сообщение типа `String` вместе с исключением:

```
throw new Exception( "Here's the story...", causalException );
```

Впоследствии вы можете получить доступ к исходному исключению методом `getCause()`. Есть и более важная возможность: если вы выводите на экран сообщение об исключении (или если его видит пользователь), то Java автоматически

предоставляет информацию об обоих исключениях вместе с их трассировками стека.

Вы можете добавлять конструктор такого рода в свои субклассы исключений (с делегированием родительскому конструктору) или же применять следующий паттерн, в котором метод `initCause()` класса `Throwable` явно определяет исходное исключение после конструирования вашего исключения, но до его выдачи:

```
try {
    // ...
} catch ( IOException cause ) {
    Exception e =
        new IOException("What we have here is a failure to communicate...");
    e.initCause( cause );
    throw e;
}
```

Иногда бывает достаточно записать информацию в журнал (в лог-файл) или выполнить некоторое действие, а затем повторно выдать исходное исключение:

```
try {
    // ...
} catch ( IOException cause ) {
    log( cause ); // Записать в журнал
    throw cause; // Выдать исключение повторно
}
```

Сужение типа при повторной выдаче исключений

До выхода Java 7, если надо было обработать несколько типов исключений в одной секции `catch`, а затем заново выдать исходное исключение, неизбежно приходилось расширить объявляемый тип исключения до того типа, который требовался для перехвата всех вариантов, или же основательно потрудиться, чтобы избежать этого. В Java 7 компилятор стал умнее. Теперь он может выполнять большую часть работы за программиста, позволяя в большинстве случаев сужать типы выдаваемых исключений до исходных типов. Лучше всего пояснить сказанное примером:

```
void myMethod() throws ZipException, SSLException
{
    try {
        // Возможная причина ZipException или SSLException
    } catch ( Exception e ) {
        log( e );
        throw e;
    }
}
```

В этом примере мы действуем примитивно: перехватываем сразу все исключения типа `Exception` в секции `catch`, чтобы сохранить данные в журнале перед

повторной выдачей исключения. До выхода Java 7 компилятор потребовал бы, чтобы секция `throws` нашего метода также объявляла о выдаче широкого типа `Exception`.

Но теперь компилятор Java в большинстве случаев способен распознать фактические типы исключений, которые могут выдаваться, позволяя указать в секции `throws` конкретный набор типов. В этом примере мы могли бы использовать и секцию `catch` с несколькими типами. Такой подход, несмотря на небольшую потерю наглядности, очень помогает более конкретно обрабатывать исключения (даже в коде, написанном до выхода Java 7) без необходимости трудоемкой переработки секций `catch`.

«Расползание» блока `try`

Команда `try` устанавливает условие для тех команд, которые она защищает: если в какой-то из них происходит исключение, то последующие команды не выполняются. Это имеет последствия для инициализации локальных переменных. Если компилятор не может определить, произойдет ли присваивание локальной переменной в блоке `try / catch`, то он не позволит использовать эту переменную. Пример:

```
void myMethod() {
    int foo;

    try {
        foo = getResults();
    }
    catch ( Exception e ) {
        ...
    }

    int bar = foo; // Ошибка компиляции: возможно, переменная foo
                  // не была инициализирована
```

В этом примере переменная `foo` не может использоваться в данном месте, потому что есть вероятность того, что ей не будет присвоено значение. Очевидное решение — переместить присваивание внутрь блока `try`:

```
try {
    foo = getResults();
    int bar = foo; // Нормально, потому что эта точка
                  // будет достигнута только при успешном выполнении
                  // предыдущего присваивания
}
catch ( Exception e ) {
    ...
}
```

Иногда это решение работает хорошо. Тем не менее при попытке дальнейшего использования переменной `bar` в методе `myMethod()` мы получим ту же проблему. Если не соблюдать осторожность, то в итоге нам придется «затолкать» все команды в блок `try`. Чтобы исправить ситуацию, можно добавить в секцию `catch` возврат из метода:

```
try {
    foo = getResults();
}
catch ( Exception e ) {
    ...
    return;
}
int bar = foo;  // Нормально, потому что эта точка
               // будет достигнута только при успешном выполнении
               // предыдущего присваивания
```

Компилятор достаточно умен, чтобы понять: если в блоке `try` произойдет ошибка, то выполнение не дойдет до точки присваивания `bar`. Поэтому компилятор позволяет обратиться к `foo`. В разных случаях ваш код будет требовать разных решений, и вы должны знать возможные варианты.

Секция `finally`

А если перед возвратом из метода через одну из секций `catch` надо сделать что-то важное? Чтобы избежать дублирования кода в каждой ветви `catch` и выполнить завершающие действия более явно, можно воспользоваться секцией `finally`. Секция `finally` добавляется после `try` и всех последующих секций `catch`. Все команды в секции `finally` гарантированно будут выполнены независимо от того, как управление вернется из `try` (с выдачей исключения или без него):

```
try {
    // Здесь что-то происходит
}
catch ( FileNotFoundException e ) {
    ...
}
catch ( IOException e ) {
    ...
}
catch ( Exception e ) {
    ...
}
finally {
    // Завершающие действия, которые выполняются всегда
}
```

В этом примере все команды в точке «Завершающие действия...» будут выполнены независимо от того, по какой причине управление передается из `try`. Если управление передается в одну из секций `catch`, то команды `finally` будут выполнены после завершения `catch`. Если ни одна из секций `catch` не обработает исключение, то команды `finally` будут выполнены до того, как исключение всплывет на следующий уровень.

Если команды в `try` будут выполнены без ошибок, а также если будет выполнена команда `return`, `break` или `continue`, то команды в секции `finally` все равно будут выполнены. Чтобы гарантировать выполнение определенных действий, можно даже использовать `try` без секций `catch`:

```
try {
    // Здесь что-то происходит
    return;
}
finally {
    System.out.println("Whoo-hoo!");
}
```

Исключение, возникшее в секции `catch` или `finally`, обрабатывается обычным образом; поиск замыкающей конструкции `try / catch` начинается за пределами проблемной команды `try`, после выполнения `finally`.

try с ресурсами

Секцию `finally` часто применяют для того, чтобы ресурсы, используемые в секции `try`, были гарантированно освобождены, независимо от того, по какой причине происходит выход из блока:

```
try {
    // Socket sock = new Socket(...);
    // Работа с sock
} catch( IOException e ) {
    ...
}
finally {
    if ( sock != null ) { sock.close(); }
}
```

В таких случаях бывает нужно освободить самые ценные ресурсы или закрыть файлы, сокеты, подключения к базам данных. Иногда ресурсы могут освобождаться автоматически в процессе уборки мусора, но это произойдет в лучшем случае в неизвестный момент будущего, а в худшем — никогда не произойдет (или не произойдет до того, как возникнет нехватка ресурсов). От таких ситуаций всегда надо защищаться. Показанный подход в целом не-

плох, но у него есть два недостатка. Во-первых, реализация этого паттерна во всем коде потребует дополнительной работы (проверки на `null`, как в нашем примере, и т. д.). Во-вторых, если вы «жонглируете» несколькими ресурсами в одном блоке `finally`, возникает риск того, что ваш код освобождения ресурсов сам выдаст исключение (например, при выполнении `close()`), а работа останется незавершенной.

В Java 7 проблема значительно упростилась благодаря новой форме секции `try`, которая называется «`try` с ресурсами». Вы можете заключить одну или несколько команд инициализации ресурсов в круглые скобки после ключевого слова `try`, и эти ресурсы будут автоматически закрыты (освобождены) после передачи управления из блока `try`:

```
try (
    Socket sock = new Socket("128.252.120.1", 80);
    FileWriter file = new FileWriter("foo");
)
{
    // Операции с sock и file
} catch ( IOException e ) {
    ...
}
```

В этом примере мы инициализируем объекты `Socket` и `FileWriter` в секции «`try` с ресурсами» и используем их в теле команды `try`. При выходе управления из команды `try` — либо после успешного завершения, либо из-за исключения — оба ресурса будут автоматически закрыты вызовом их метода `close()`. Ресурсы закрываются в порядке, обратном порядку их конструирования, что учитывает возможные зависимости между ними. Такое поведение поддерживается любым классом, реализующим интерфейс `AutoCloseable` (который в настоящий момент поддерживается более чем ста разными встроенными классами). Метод `close()` этого интерфейса должен освобождать все ресурсы, связанные с объектом, и вы можете легко реализовать его в своих собственных классах. При использовании «`try` с ресурсами» вам не придется добавлять специальный код для закрытия файла или сокета; это делается за вас автоматически.

Другая проблема, которую решает «`try` с ресурсами», — это уже упомянутая неприятная ситуация с выдачей исключения во время операции закрытия. Если в предыдущем примере, где для освобождения ресурсов использовалась секция `finally`, возникнет исключение в методе `close()`, то оно будет выдано из этой точки, а исходное исключение из тела `try` окажется полностью потерянным. Но при использовании «`try` с ресурсами» исходное исключение сохранится. Если вслед за исключением в теле `try`, во время последующих операций автоматического закрытия, возникнет одно или несколько исключений, то именно исходное исключение всплывет из тела `try` к вызывающей стороне. Рассмотрим пример:

```
try (  
    Socket sock = new Socket("128.252.120.1", 80); // Возможное исключение 3  
    FileWriter file = new FileWriter("foo"); // Возможное исключение 2  
)  
{  
    // Операции с sock и file // Возможное исключение 1  
}
```

После начала `try`, если в точке 1 происходит исключение, Java пытается закрыть оба ресурса в обратном порядке, что приводит к возможным исключениям в точках 2 и 3. В этом случае вызывающий код все равно получит исключение 1. Впрочем, исключения 2 и 3 не пропадают; они просто «подавляются» и могут быть найдены методом `getSuppressed()` класса `Throwable` при обработке исключения, выданного вызывающей стороне. Этот метод возвращает массив всех подавленных исключений.

Обработка исключений и быстроедействие

Виртуальная машина Java устроена таким образом, что защита от выданного исключения (с помощью `try`) всегда «бесплатна». Она не создает никакой дополнительной нагрузки, влияющей на выполнение вашего кода. Тем не менее само по себе исключение не является «бесплатным». Когда в выполняемом коде возникает исключение, Java ищет подходящий блок `try / catch` и занимается другими длительными операциями.

Это означает, что исключения надо выдавать только в действительно исключительных ситуациях. Старайтесь не использовать исключения в нормальных ситуациях, особенно если для вас важно быстроедействие. Например, при проектировании цикла лучшим решением может быть небольшая проверка в каждой итерации, позволяющая избежать многократно возникающих исключений. Но если исключение выдается только один раз из миллиарда, то вы, вероятно, предпочтете отказаться от проверочного кода и не беспокоиться о затратах на выдачу исключения. Помните, что исключения предназначены для ненормальных ситуаций, а не для обычных и ожидаемых условий (таких, как конец файла).

Проверочные утверждения

Проверочное утверждение (assertion) — это простая проверка некоторого условия во время выполнения вашего приложения. Проверочные утверждения нужны для контрольных проверок вашего кода в тех местах, где вы уверены, что при правильном поведении программы гарантируется соблюдение некоторых условий. Проверочные утверждения отличаются от других проверок, потому что они проверяют условия, которые никогда не должны нарушаться на логическом уровне:

если проверка завершается неудачей, значит, приложение работает некорректно; обычно в таких случаях оно аварийно завершается с выдачей подходящего сообщения об ошибке. Проверочные утверждения напрямую поддерживаются языком Java, и их можно отключать во время выполнения приложения, чтобы избавиться от вызванных ими потерь быстродействия.

Применение проверочных утверждений для проверки правильности поведения приложения — это простое, но мощное средство контроля качества кода. Проверочные утверждения заполняют пробел между теми аспектами программного продукта, которые могут автоматически проверяться компилятором, и теми, которые чаще контролируются «модульными тестами» и тестированием с участием человека. Проверочные утверждения (если они включены) проверяют предположения относительно поведения программы и превращают их в гарантии. Если у вас уже есть опыт программирования, возможно, вы встречали конструкции следующего вида¹:

```
if ( !condition )  
    throw new AssertionError("fatal error: 42");
```

Проверочное утверждение в Java эквивалентно этому примеру, но оно выполняется ключевым словом `assert`. При вызове передается логическое условие и необязательное выражение. Если условие не выполняется, то выдается исключение `AssertionError`, что обычно приводит к аварийному завершению приложения Java.

Необязательное выражение может давать в результате вычисления примитивное значение или объектный тип. В любом случае оно имеет только одно предназначение: преобразование в строку и вывод для пользователя, если проверка завершается неудачей. Чаще всего используется конкретизированное строковое сообщение. Несколько примеров:

```
assert false;  
assert ( array.length > min );  
assert a > 0 : a // Выводит значение a  
assert foo != null : "foo is null!" // Выводит сообщение "foo is null"
```

При неудаче первые два проверочных утверждения выводят только общее сообщение, тогда как третье выводит значение переменной, а последнее — сообщение `"foo is null"`.

Еще раз: важное преимущество проверочных утверждений заключается не в том, что они компактнее эквивалентных условий `if`, а в том, что их можно включать

¹ Если у вас уже имеется опыт программирования, надеемся, вы не используете такие невразумительные сообщения об ошибках! Чем понятнее и содержательнее ваши сообщения, тем лучше.

или отключать при запуске приложения. Отключение проверочных утверждений означает, что не только не вычисляются проверяемые ими условия, но и само присутствие этих строк в вашем коде не приводит к снижению быстродействия (если не считать памяти файлов классов при их загрузке).

Включение и отключение проверочных утверждений

Проверочные утверждения включаются и выключаются перед запуском приложения. Отключенные проверочные утверждения остаются в файлах классов, но не выполняются и не отнимают времени. Проверочные утверждения можно включать и отключать для всего приложения, или на уровне отдельных пакетов, или даже на уровне классов. По умолчанию все проверочные утверждения отключены. Чтобы включить их для всего приложения, используйте команду `java` с флагом `-ea` (или `-enableassertions`):

```
$ java -ea MyApplication
```

Чтобы включить проверочные утверждения для конкретного класса, укажите после флага имя класса:

```
$ java -ea:com.oreilly.examples.Myclass MyApplication
```

Чтобы включить проверочные утверждения только для некоторых пакетов, укажите имя пакета с завершающим многоточием (...):

```
$ java -ea:com.oreilly.examples... MyApplication
```

Когда вы включаете проверочные утверждения для пакета, Java также включает их для всех подчиненных имен пакетов (например, `com.oreilly.examples.text`). Но вы можете действовать более избирательно, используя флаг `-da` (или `-disableassertions`), отключающий проверочные утверждения для отдельных пакетов или классов. Все это можно объединять в произвольные группы, например, так:

```
$ java -ea:com.oreilly.examples... \
      -da:com.oreilly.examples.text \
      -ea:com.oreilly.examples.text.MonkeyTypewriters MyApplication
```

В этом примере проверочные утверждения включаются для всего пакета `com.oreilly.examples`, кроме пакета `com.oreilly.examples.text`, а также только для одного класса `MonkeyTypewriters` из этого пакета.

Использование проверочных утверждений

Проверочное утверждение устанавливает правило, которое не должно изменяться в вашем коде и которое без проверочного утверждения осталось бы непроверен-

ным. Проверочное утверждение может использоваться для обеспечения безопасности в любых местах, где вы хотите проверить свои предположения относительно некоторого поведения программы, которое не может проверяться компилятором.

Типичная ситуация, в которой уместно применение проверочного утверждения, — проверка нескольких условий или значений, одно из которых обязательно должно присутствовать. В таком случае невыполнение проверочного утверждения как поведения по умолчанию указывает на то, что где-то в коде есть ошибки. Допустим, у вас имеется значение `direction`, которое всегда должно содержать константу `LEFT` или `RIGHT`:

```
if ( direction == LEFT )
    doLeft();
else if ( direction == RIGHT )
    doRight()
else
    assert false : "bad direction";
```

То же относится к секции `default` команды `switch`:

```
switch ( direction ) {
    case LEFT:
        doLeft();
        break;
    case RIGHT:
        doRight();
        break;
    default:
        assert false;
}
```

В общем случае проверочные утверждения не следует использовать для проверки действительности аргументов методов, потому что это поведение должно быть частью вашего приложения, а не обычной проверкой качества, которую можно отключить. Действительность входных данных метода называется его *предусловием*, и при его нарушении обычно должно выдаваться исключение; таким образом предусловия повышаются до составляющей «контракта» метода с пользователем. С другой стороны, проверка правильности результатов методов перед их возвращением — хорошая мысль; такие проверки называются *постусловиями*.

Иногда определение того, что является или не является предусловием, зависит от точки зрения. Например, когда метод используется во внутренней реализации класса, предусловия могут уже гарантироваться вызывающими его методами. Вероятно, открытые методы класса должны выдавать исключения при нарушении их предусловий, но скрытые (приватные) методы могут использовать проверочные утверждения, потому что их вызывающие методы всегда тесно связаны с ними, а их код должен соблюдать правильное поведение.

Журнальный API

Пакет `java.util.logging` предоставляет чрезвычайно гибкую и простую в использовании основу для вывода системной информации, сообщений об ошибках и детализированной трассировки (отладочных данных). С пакетом `logging` вы можете устанавливать фильтры для выбора записываемых в журнал сообщений, направлять их вывод в один или несколько приемников (включая файлы и сетевые службы) и форматировать сообщения для потребителей.

Но что самое важное, большая часть базовой конфигурации `logging` может настраиваться извне во время выполнения приложения — с помощью файла настроек или внешней программы. Например, задавая нужные настройки во время выполнения приложения, можно указать, что сообщения в журнале должны сохраняться в указанном файле в формате XML, а также направляться на системную консоль в удобочитаемом виде. Кроме того, для каждого приемника можно задать уровень (или приоритет) выводимых сообщений; сообщения, не достигающие некоторого порога важности, будут теряться. Соблюдая необходимые соглашения в вашем коде, можно даже настраивать уровни ведения журнала для конкретных частей вашего приложения; это позволяет включать подробный вывод для отдельных пакетов и классов без перегрузки избыточным выводом. Журнальным API Java даже можно управлять в удаленном режиме через API Java Management Extensions MBean.

Общие сведения

Любой хороший журнальный API должен руководствоваться по крайней мере двумя основными принципами. Во-первых, он не должен существенно замедлять работу приложений, то есть препятствовать свободному использованию сообщений разработчиком. Как и в случае с проверочными утверждениями Java, отключенные журнальные сообщения не должны потреблять сколько-нибудь значительного времени. Это означает, что наличие команд вывода в журнал не должно приводить к снижению производительности, если эти команды отключены. Во-вторых, хотя некоторым пользователям может потребоваться расширенная функциональность и возможность детальной настройки, журнальный API должен иметь простой режим использования. Он должен быть достаточно удобен, чтобы разработчики, которым вечно не хватает времени, могли пользоваться им вместо старого метода `System.out.println()`. Журнальный API в языке Java предоставляет и упрощенную модель, и множество дополнительных, чрезвычайно удобных методов¹.

¹ Тем, кому возможностей журнального API окажется недостаточно, рекомендуем Apache Log4j 2 (<https://logging.apache.org/log4j/2.x>) и SLF4J (Simple Logging Facade for

Протоколировщики

В системе журнального вывода центральное место занимает *протоколировщик* — экземпляр класса `java.util.logging.Logger`. Как правило, это единственный класс, с которым вам надо иметь дело в вашем коде. Протоколировщик создается статическим методом `Logger.getLogger()`, которому в аргументе передается имя протоколировщика. Имена формируют иерархию, на вершине которой находится глобальный корневой протоколировщик, а ниже находится дерево его потомков. Эта иерархия позволяет наследовать конфигурацию в отдельных частях дерева, чтобы журнальный вывод мог автоматически настраиваться для разных частей вашего приложения. По действующим соглашениям в каждом крупном классе или пакете используется отдельный экземпляр протоколировщика, а в качестве его имени используется имя пакета и (или) имя класса с точками в качестве разделителей. Пример:

```
package com.oreilly.learnjava;
public class Book {
    static Logger log = Logger.getLogger("com.oreilly.learnjava.Book");
```

Протоколировщик предоставляет широкий диапазон методов для сохранения сообщений в журнале; одни получают очень подробную информацию, другие — только одну строку для простоты использования. Пример:

```
log.warning("Disk 90% full.");
log.info("New user joined chat room.");
```

Методы класса протоколировщика будут подробно рассмотрены ниже. Имена `warning` и `info` — примеры уровней вывода; всего определено семь уровней, от `SEVERE` (верхний) до `FINEST` (нижний). Возможность различать журнальные сообщения позволяет выбрать уровень информации, которую вы хотите видеть во время выполнения. Вместо того чтобы сохранять все подряд и сортировать позднее (это плохо влияет на скорость работы), вы можете указать, какие именно сообщения должны генерироваться. Уровни вывода рассматриваются в следующем разделе.

Также стоит упомянуть, что для очень простых приложений или экспериментов протоколировщик для имени `global` содержится в статическом поле `Logger.global`. Вы можете использовать его как альтернативу классическому выводу `System.out.println()`:

```
Logger.global.info("Doing foo...")
```

Java) (<http://www.slf4j.org>), позволяющие еще точнее настраивать уровни журнального вывода.

Обработчики

Протоколировщики представляют клиентский интерфейс к системе журнального вывода, но непосредственная работа по выводу сообщений в приемники (например, в файлы или на консоль) выполняется объектами-обработчиками. С каждым протоколировщиком может быть связан один или несколько таких объектов (субклассов класса `Handler`), включая обработчики, заранее определенные в журнальном API: `ConsoleHandler`, `FileHandler`, `StreamHandler` и `SocketHandler`. Каждый обработчик знает, как доставлять сообщения своему приемнику.

`ConsoleHandler` используется конфигурацией по умолчанию для вывода сообщений в командной строке или в системной консоли. `FileHandler` может направлять вывод в файлы с заданной схемой имен, с автоматической ротацией файлов при их заполнении. Другие обработчики отправляют данные в потоки и сокеты соответственно. Также существует дополнительный обработчик `MemoryHandler`, который может хранить некоторое количество журнальных сообщений в памяти. `MemoryHandler` использует циклический буфер, который хранит определенное количество сообщений до того момента, как будет инициирована их отправка другому назначенному обработчику.

Как было сказано выше, протоколировщики могут настраиваться для использования одного или нескольких обработчиков. Протоколировщики также отправляют сообщения вверх по дереву к каждому из обработчиков родительского протоколировщика. В простейшей конфигурации это означает, что все сообщения в конечном итоге будут распределяться обработчиками корневого протоколировщика. Вскоре мы покажем, как настроить вывод с использованием стандартных обработчиков для консоли, файлов и т. д.

Фильтры

Прежде чем передавать сообщение своим или родительским обработчикам, протоколировщик сначала проверяет, достаточен ли уровень вывода для продолжения. Если сообщение не соответствует необходимому уровню, оно отбрасывается в источнике. Помимо проверки уровня, можно реализовать произвольную фильтрацию сообщений; для этого следует создать классы `Filter` для проверки сообщения перед обработкой. Класс `Filter` может быть применен к протоколировщику извне (так же, как и уровень вывода, обработчики и форматировщики, о которых будет рассказано в следующем разделе). Класс `Filter` также можно связать с конкретным экземпляром `Handler` для фильтрации записей на стадии вывода, в отличие от фильтрации в источнике.

Форматировщики

Во внутреннем представлении сообщения передаются в нейтральном формате, с включением всей предоставленной информации об источнике. Только после

применения обработчика сообщения форматируются для вывода экземпляром объекта `Formatter`. В журнальном API есть два базовых форматировщика: `SimpleFormatter` и `XMLFormatter`. «Простой» `SimpleFormatter` используется по умолчанию для консольного вывода. Он производит короткие, понятные для человека сводки журнальных сообщений. `XMLFormatter` кодирует все подробности сообщения в формате записи XML. Схемы DTD для этого формата доступны на сайте Oracle: <http://www.oracle.com/webfolder/technetwork/jsc/dtd/index.html>.

Уровни вывода

В табл. 6.1 перечислены уровни вывода в порядке убывания значимости.

Таблица 6.1. Уровни вывода в журнальном API

Уровень	Смысл
SEVERE	Фатальный сбой приложения
WARNING	Уведомление о потенциальной проблеме
INFO	Сообщения, представляющие интерес для конечного пользователя
CONFIG	Подробная информация о конфигурации системы для администратора
FINE, FINER, FINEST	Трассировочные данные для разработчика (с последовательным увеличением уровня детализации)

Все уровни делятся на три категории: для конечного пользователя, для администратора и для разработчика. По умолчанию приложения часто ограничиваются сообщениями уровня `INFO` и выше (`INFO`, `WARNING` и `SEVERE`). Эти уровни обычно видны конечным пользователям, а сообщения, выводимые на этих уровнях, должны быть пригодными для общего применения. Иначе говоря, они должны быть сформулированы так, чтобы быть понятными (или хотя бы осмысленными) для рядовых пользователей приложения. Часто такие сообщения выводятся на системную консоль или во временном диалоговом окне.

Уровень `CONFIG` должен использоваться для относительно статической, но подробной системной информации, которая должна быть рассчитана на администратора или специалиста, устанавливающего программу. Такие сообщения могут включать информацию об установленных программных модулях, характеристиках системы и параметрах конфигурации. Эти подробности важны, но для конечных пользователей они, скорее всего, не представляют интереса.

Уровни `FINE`, `FINER` и `FINEST` предназначены для разработчиков и других специалистов, разбирающихся во внутренней архитектуре приложения. Эти уровни

должны использоваться для трассировки приложения на последовательно возрастающих уровнях детализации. Вы можете определить для них собственную интерпретацию. В следующем разделе будет представлена одна из возможных схем.

Простой пример

В следующем (откровенно говоря, искусственном) примере мы используем все уровни вывода для экспериментов с конфигурацией журнального вывода. Хотя последовательность сообщений выглядит бессмысленно, эти тексты типичны для сообщений такого типа.

```
import java.util.logging.*;

public class LogTest {
    public static void main(String argv[])
    {
        Logger logger = Logger.getLogger("com.oreilly.LogTest");

        logger.severe("Power lost - running on backup!");
        logger.warning("Database connection lost, retrying...");
        logger.info("Startup complete.");
        logger.config("Server configuration: standalone, JVM version 1.5");
        logger.fine("Loading graphing package.");
        logger.finer("Doing pie chart");
        logger.finest("Starting bubble sort: value =" + 42);
    }
}
```

Этот пример не особенно содержателен. Сначала мы запрашиваем экземпляр `logger` для своего класса статическим методом `Logger.getLogger()`, передавая ему имя класса. По действующим соглашениям должно использоваться полное имя класса, поэтому мы будем считать, что наш класс принадлежит пакету `com.oreilly`.

Теперь запустите программу `LogTest`. На системной консоли должен появиться вывод, который выглядит примерно так:

```
Jan 6, 2019 3:24:36 PM LogTest main
SEVERE: Power lost - running on backup!
Jan 6, 2019 3:24:37 PM LogTest main
WARNING: Database connection lost, retrying...
Jan 6, 2019 3:24:37 PM LogTest main
INFO: Startup complete.
```

Мы видим сообщения `INFO`, `WARNING` и `SEVERE` с указанием даты, временной метки и имени класса и метода (`LogTest`, `main`), из которого они поступили.

Обратите внимание: сообщений более низкого уровня нет. Дело в том, что по умолчанию обычно выбирается уровень вывода `INFO`; это означает, что выводятся только сообщения с уровнем `INFO` и выше. Также обратите внимание, что вывод направляется на системную консоль, а не в журнальный файл; этот приемник также используется по умолчанию. А теперь мы опишем, где выбираются эти настройки по умолчанию и как переопределить их во время выполнения.

Конфигурирование журнального API

Как было сказано во вводной части, очень важной особенностью журнального API является возможность настройки детализации вывода на стадии выполнения с помощью внешнего файла или специального приложения. Конфигурация вывода по умолчанию хранится в файле `jre/lib/logging.properties`, который находится в рабочем каталоге Java. Это стандартный конфигурационный файл; мы уже упоминали о таких файлах в этой главе.

У этого файла простой формат. Вы можете вносить в него изменения, но это не обязательно. Вместо этого вы можете задать собственный файл для настройки журнального вывода в каждом конкретном случае, например, таким образом:

```
$ java -Djava.util.logging.config.file=myfile.properties
```

В этой командной строке `myfile` — ваш конфигурационный файл с директивой, которая будет описана ниже. Если вы хотите использовать этот файл на постоянной основе, укажите его имя в соответствующей записи с помощью Java Preferences API. Можно даже пойти еще дальше и вместо файла с настройками указать класс, ответственный за подготовку всей конфигурации журнального вывода, но мы эту возможность рассматривать не будем.

Очень простой конфигурационный файл может выглядеть так:

```
# Set the default logging level
.level = FINEST
# Direct output to the console
handlers = java.util.logging.ConsoleHandler
```

Здесь уровень вывода по умолчанию назначается для всего приложения при помощи параметра с именем `.level` (обратите внимание на точку). Также параметр `handlers` указывает, что при выводе должен использоваться экземпляр `ConsoleHandler` (как в конфигурации по умолчанию), чтобы сообщения выводились на консоль. Если вы снова запустите приложение, указав этот файл для определения конфигурации журнала, вы снова увидите все сообщения.

Впрочем, это только начало. Перейдем к более сложной конфигурации:

```
# Set the default logging level
.level = INFO

# Output to file and console
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Configure the file output
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

# Configure the console output
java.util.logging.ConsoleHandler.level = WARNING

# Levels for specific classes
com.oreilly.LogTest.level = FINEST
```

В этом примере настраиваются два обработчика: экземпляр `ConsoleHandler` с уровнем вывода `WARNING` и экземпляр `FileHandler`, направляющий вывод в XML-файл. Файловый обработчик настроен для регистрации сообщений на уровне `FINEST` (все сообщения) и для ротации файлов через каждые 25 000 строк, с хранением максимум 4 файлов.

Имя файла определяется параметром `pattern`. Символы `/` в имени файла при необходимости автоматически локализируются в `\`. Специальная последовательность `%h` обозначает домашний каталог пользователя. Для ссылок на системный временный каталог может использоваться последовательность `%t`. При возникновении конфликтов между именами файлов после точки автоматически присоединяется число (начиная с нуля). Также можно воспользоваться последовательностью `%i`, чтобы обозначить позицию вставки уникального числа в имя. Аналогичным образом при ротации файлов после точки в конце присоединяется номер. Идентификатор `%g` управляет позицией вставки номера ротации.

В нашем примере используется класс `XMLFormatter`. Также можно было воспользоваться классом `SimpleFormatter` для отправки того же простого вывода на консоль. `ConsoleHandler` также позволяет задать любой форматировщик на ваш выбор при помощи параметра `formatter`.

Наконец, ранее мы упоминали о том, что можно управлять уровнями вывода для отдельных частей приложения. Для этого надо задать параметры протоколировщиков приложения с указанием их иерархических имен:

```
# Levels for specific logger (class) names
com.oreilly.LogTest.level = FINEST
```

Здесь уровень вывода назначается только для нашего тестового протоколировщика. Параметры соответствуют иерархии, поэтому уровень вывода для всех классов пакета `oreilly` можно задать следующей командой:

```
com.oreilly.level = FINEST
```

Уровни вывода задаются в порядке их чтения из конфигурационного файла, поэтому самые общие параметры должны быть указаны в начале. Уровни, заданные для обработчиков, позволяют файловому обработчику фильтровать только сообщения, поставляемые протоколировщиками. Следовательно, назначение файловому обработчику уровня `FINEST` не «оживит» сообщения, подавленные протоколировщиком, которому назначен уровень `SEVERE` (от этого протоколировщика до обработчика доберутся только сообщения `SEVERE`).

Протоколировщик

В нашем примере использовались семь вспомогательных методов, имена которых соответствовали разным уровням вывода. Также есть три группы общих методов, которые могут использоваться для передачи более подробной информации. Наиболее важны следующие:

```
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 первом аргументе этих методов передается статический идентификатор уровня вывода из класса `Level`, за которым следует параметр, массив или тип исключения. Идентификатор уровня имеет вид `Level.SEVERE`, `Level.WARNING`, `Level.INFO` и т. д.

Кроме этих четырех методов, также существуют вспомогательные методы с именами `entering()`, `exiting()` и `throwing()`, которые могут использоваться разработчиками для вывода подробных трассировочных данных.

Быстродействие

Во вводной части мы упоминали, что быстродействие является приоритетной целью журнального API. В частности, мы упоминали, что журнальные сообщения фильтруются в источнике, при этом уровни вывода используются для раннего отсека сообщения. При этом экономится значительная часть затрат на их обработку. Однако фильтрация не может предотвратить некоторые виды подготовительных операций, которые могут выполняться перед вызовом. А именно, поскольку мы передаем данные журнальным методом, на практике часто

конструируются подробные сообщения или объекты преобразуются в строки, передаваемые в аргументах. Такие операции часто сопряжены со значительными ресурсами. Чтобы избежать конструирования лишних строк, затратные операции надо «обернуть» в проверку условия методом `isLoggable()` класса `Logger` чтобы уточнить, должны ли эти операции выполняться, например:

```
if ( log.isLoggable( Level.CONFIG ) ) {  
    log.config("Configuration: "+ loadExpensiveConfigInfo() );  
}
```

Исключения в реальном мире

Применение исключений для обработки ошибок в Java значительно упрощает написание кода, защищенного от возможных ошибок. Компилятор заставляет вас заранее продумывать проверяемые исключения. Безусловно, время от времени будут возникать непроверяемые исключения, но проверочные утверждения помогут вам заранее предусмотреть эти проблемы, а если повезет, то и предотвратить ошибки.

Синтаксис «try с ресурсами», появившийся в Java 7, помогает разработчикам поддерживать чистоту кода и «поступать правильно» при взаимодействии с ограниченными системными ресурсами, такими как файлы и сетевые подключения. Как упоминалось в начале главы, в других языках тоже предусмотрены различные средства для решения подобных проблем. Язык Java старается помочь вам тщательно продумать проблемы, которые могут возникнуть в вашем коде. И чем больше вы трудитесь над разрешением этих проблем, тем стабильнее работает ваше приложение и тем лучшие впечатления остаются у пользователей.

Для выявления самых коварных, «незаметных» ошибок, не приводящих к сбою вашего приложения, Java предоставляет пакет журналирования `java.util.logging`, позволяющий отслеживать причины таких ошибок. Вы всегда сможете регулировать объем выводимых в журнал данных, сохраняя разумную скорость работы приложения.

До сих пор многие из наших примеров были простыми и не требовали хитроумного контроля ошибок. Не сомневайтесь: мы еще займемся более интересным кодом и многими ситуациями, в которых необходима обработка исключений. В следующих главах будут рассмотрены такие темы, как многопоточное программирование и работа в сети. В этих областях бывает множество ситуаций, которые могут создать проблемы при выполнении кода, — например, выход из-под контроля сложных вычислений или потеря подключения Wi-Fi. Скоро вы сможете попробовать (с помощью команды `try`) все эти новые приемы работы с ошибками и исключениями.

Коллекции и обобщения

Когда вы начнете применять свои растущие знания об объектах для решения все более интересных задач, у вас будет возникать один и тот же вопрос. Как хранить данные, которыми вы оперируете при решении этих задач? Конечно, для этого есть переменные всех типов, но в дополнение к ним вам понадобятся более сложные и вместительные хранилища. Массивы, которые мы рассматривали в одноименном разделе, с. 144, могут стать отправной точкой, но у них есть некоторые ограничения. В этой главе мы объясним, как осуществляется более эффективный и гибкий доступ к большим объемам данных. В этом вам поможет API коллекций Java. Вы также узнаете, как сохранять в коллекциях (то есть в больших хранилищах) данные разных типов и работать с ними почти так же, как с отдельными значениями в переменных. Для этого в Java есть обобщения (generics, дженерики), которые рассматриваются в разделе «Ограничения типов», с. 242.

Коллекции

Коллекции — это структуры данных, лежащие в основе всех видов программирования. Любую обозначенную вами группу объектов вы можете считать своего рода коллекцией. Например, на базовом уровне Java поддерживает коллекции в виде массивов. Но массивы статичны. Из-за фиксированной длины они неудобны для хранения объектов в таких группах, которые увеличиваются и уменьшаются во время работы приложения. Кроме того, в массивах трудно отражать абстрактные взаимосвязи между объектами. В ранних версиях Java было только два простейших класса для частичного решения этих проблем: класс `java.util.Vector`, представляющий динамический список объектов, и класс `java.util.Hashtable`, представляющий карту (ассоциативный массив) из пар «ключ — значение». Но теперь в Java реализован полноценный подход к коллекциям в виде фреймворка коллекций (Collections Framework). Оба старых класса все еще поддерживаются, но они были модифицированы для этого фреймворка (немного странным образом) и, как правило, уже не используются.

Коллекции очень просты по сути, но они относятся к самым мощным средствам во всех языках программирования. Они помогают программистам создавать такие структуры данных, которые критически важны для решения сложных проблем. В компьютерных науках уделяют много внимания тому, как эффективно выполнять некоторые типы алгоритмов с помощью коллекций. Если у вас есть эти инструменты и вы умеете работать с ними, то ваш код становится заметно компактнее и быстрее. И вам не приходится заново «изобретать велосипед».

Первая версия фреймворка коллекций вышла с двумя большими недостатками. Во-первых, коллекции были нетипизованными, то есть работали только с недифференцированными объектами `Object` вместо конкретных типов (таких, как `Date` или `String`). Поэтому при каждом извлечении объекта из коллекции приходилось выполнять преобразование типов, что явно противоречило ключевой идее Java — безопасности типов на стадии компиляции. На практике это решение не вызывало больших проблем, но оказалось громоздким и неудобным. Во-вторых, коллекции могли состоять только из объектов, а примитивные типы не поддерживались. Поэтому каждый раз, когда требовалось внести в коллекцию число или другой примитивный тип, его приходилось предварительно сохранять в классе-обертке, а после извлечения — распаковывать обратно. Из-за этих недостатков код для работы с коллекциями получался трудным для понимания и ненадежным.

Все стало гораздо лучше, когда во фреймворке коллекций появились обобщенные типы и механизм автоматического преобразования примитивных типов. Обобщенные типы (подробнее см. «Ограничения типов», с. 242) дают программисту уверенность в совершенно безопасной типизации при работе с коллекциями. А благодаря автоматической упаковке и распаковке примитивных типов их можно использовать в коллекциях точно так же, как объекты. Эти новые инструменты делают код намного компактнее и безопаснее, и теперь они есть во всех классах коллекций Java, как вы вскоре увидите.

Фреймворк коллекций основан на наборе интерфейсов из пакета `java.util`. Все они делятся на две ветви с иерархической структурой. Одна из них унаследована от интерфейса `Collection`, который (как и его потомки) определяет контейнер, способный хранить другие объекты. Вторая, отдельная ветвь основана на интерфейсе `Map`, определяющем группу пар «ключ — значение»; ключи в таких парах нужны, чтобы легко находить нужные значения.

Интерфейс Collection

Предком всех коллекций первой ветви является интерфейс с подходящим именем `Collection`. Он служит контейнером, *элементами* которого являются объекты. Контейнер не содержит информации о том, как следует хранить эти объекты; например, он «не знает» о том, разрешены ли дубликаты и упорядочены ли объекты каким-то образом. Все подобные детали определяются дочерними

интерфейсами. Тем не менее интерфейс `Collection` определяет ряд базовых методов, общих для всех этих коллекций:

`public boolean add(элемент)`

Метод добавляет заданный элемент в коллекцию. Если эта операция завершится успешно, метод возвращает `true`. Если объект уже есть в коллекции, а дубликаты в ней не разрешены, метод возвращает `false`. Некоторые коллекции доступны только для чтения; при вызове этого метода они выдают исключение `UnsupportedOperationException`.

`public boolean remove(элемент)`

Метод удаляет заданный элемент из коллекции. Как и метод `add()`, он возвращает `true` при удалении объекта из коллекции. Если этого объекта нет в коллекции, он возвращает `false`. Коллекции, доступные только для чтения, при вызове этого метода тоже выдают исключение `UnsupportedOperationException`.

`public boolean contains(элемент)`

Метод возвращает `true`, если в коллекции есть заданный элемент.

`public int size()`

Метод возвращает количество элементов в коллекции.

`public boolean isEmpty()`

Метод возвращает `true`, если коллекция не содержит элементов.

`public Iterator iterator()`

Метод проверяет все элементы в коллекции. Он возвращает итератор (`Iterator`) — объект, который можно использовать для перебора всех элементов коллекции. Итераторы подробно рассматриваются в следующем разделе.

Кроме перечисленных, есть методы `addAll()`, `removeAll()` и `containsAll()`, которые принимают в качестве аргумента другой (сравниваемый) объект `Collection` и, соответственно, добавляют, удаляют и проверяют элементы заданной коллекции.

Разновидности коллекций

Интерфейс `Collection` имеет три дочерних интерфейса. `Set` («множество») определяет коллекцию, в которой дубликаты недопустимы. В коллекции `List` («список») элементы следуют в определенном порядке. Интерфейс `Queue` («очередь») определяет буфер для объектов, в котором находящийся в самом начале, то есть головной элемент должен быть обработан в первую очередь.

Интерфейс Set

Интерфейс `Set` не содержит других методов, кроме унаследованных от `Collection`. Он просто выполняет правило запрета дубликатов. Если вы попытаетесь добавить элемент, уже существующий в коллекции `Set`, то метод `add()` вернет `false`. Субинтерфейс `SortedSet` позволяет хранить элементы в определенном порядке: как отсортированный список без дубликатов. Для извлечения подмножеств (тоже отсортированных) есть методы `subSet()`, `headSet()` и `tailSet()`. Они принимают в качестве аргументов один или два элемента, которые являются границами подмножества. Для доступа к первому и последнему элементу есть методы `first()` и `last()`, а метод `comparator()` предоставляет доступ к объекту, который используется для сравнения элементов (см. также раздел «Метод `sort()`», с. 260).

В Java 7 появился интерфейс `NavigableSet`, который расширяет `SortedSet` и добавляет методы для поиска ближайшего (по порядку сортировки) большего или меньшего значения по отношению к заданному. Этот интерфейс может быть эффективно реализован при использовании таких алгоритмов, как списки с пропусками (`skip lists`), ускоряющие поиск упорядоченных элементов.

Интерфейс List

Следующий дочерний интерфейс `Collection` — `List`. Он представляет упорядоченную коллекцию, сходную с массивом, но содержащую методы для изменения позиций элементов в списке:

```
public boolean add( E элемент )
```

Метод добавляет заданный элемент в конец списка.

```
public void add( int индекс , E элемент )
```

Метод вставляет заданный объект в заданную позицию списка. Если позиция меньше нуля или больше длины списка, то выдается исключение `IndexOutOfBoundsException`. Элемент, который ранее находился в заданной позиции, и все последующие элементы сдвигаются на одну позицию вверх.

```
public void remove( int индекс )
```

Метод удаляет элемент в заданной позиции. Все последующие элементы сдвигаются на одну позицию вниз.

```
public E get( int индекс )
```

Метод возвращает элемент в заданной позиции.

```
public Object set( int индекс , E элемент )
```

Метод заменяет элемент в заданной позиции заданным элементом. В этой позиции уже должен присутствовать элемент, в противном случае выдается исключение `IndexOutOfBoundsException`.

Под типом `E` в этих методах понимается параметризованный тип элемента класса `List`.

Интерфейс `Queue`

Интерфейс `Queue` представляет коллекцию, которая служит очередью (буфером) для элементов. Она сохраняет порядок вставленных в нее элементов, и в ней предусмотрено понятие «начального» («головного») элемента. Очереди могут быть двух видов: «первым пришел, первым вышел» (FIFO) или «последним пришел, первым вышел» (LIFO) в зависимости от реализации:

```
public boolean offer( E элемент ), public boolean add( E элемент )
```

Метод `offer()` пытается поместить элемент в очередь, возвращая `true` в случае успеха. Разные типы `Queue` могут устанавливать разные ограничения для типов элементов (а также для длины очереди). В отличие от метода `add()`, унаследованного от `Collection`, этот метод вместо исключения возвращает логическое значение, чтобы сообщить, что элемент не может быть принят.

```
public E poll(), public E remove()
```

Метод `poll()` удаляет элемент в начале очереди и возвращает его. В отличие от метода `remove()`, унаследованного от `Collection`, в случае пустой очереди он возвращает `null` вместо исключения.

```
public E peek()
```

Метод возвращает начальный (головной) элемент, не удаляя его из очереди. Если очередь пуста, он возвращает `null`.

Интерфейс `Map`

В составе фреймворка коллекций также есть `java.util.Map` — карта (map), то есть коллекция пар «ключ — значение» (также используются термины «словарь» и «ассоциативный массив»). В карте элементы сохраняются и читаются по значениям ключей; карты хорошо подходят для реализации таких структур, как кэши и простейшие базы данных. При сохранении значения в карте с ним связывается объект ключа. Когда вам требуется прочесть значение, карта использует для этого ключ.

Обобщенный тип `Map` параметризуется двумя типами: типом ключа и типом значения. В следующем фрагменте используется коллекция `HashMap` — эффективная, но неупорядоченная реализация карты, которая будет рассмотрена далее:

```
Map<String, Date> dateMap = new HashMap<String, Date>();
dateMap.put( "today", new Date() );
Date today = dateMap.get( "today" );
```


В старом коде карты были несовершенными: они просто ассоциировали объекты типа `Object` с объектами типа `Object`, а чтение значений требовало соответствующих преобразований типов.

Базовые операции `Map` достаточно просты. В следующих методах тип `K` обозначает тип параметра-ключа, а тип `V` — тип параметра-значения:

```
public V put( K ключ , V значение )
```

Метод добавляет в карту заданную пару «ключ — значение». Если карта уже содержит значение с заданным ключом, то старое значение заменяется и возвращается как результат операции.

```
public V get( K ключ )
```

Метод читает из карты значение, соответствующее ключу.

```
public V remove( K ключ )
```

Метод удаляет из карты значение, соответствующее ключу. Удаленное значение возвращается как результат операции.

```
public int size()
```

Метод возвращает количество пар «ключ — значение» в карте.

Для получения всех ключей или всех значений в карте есть следующие методы:

```
public Set keySet()
```

Метод возвращает объект `Set`, содержащий все ключи из карты.

```
public Collection values()
```

Метод используется для получения всех значений из карты. Полученный объект `Collection` может содержать дубликаты.

```
public Set entrySet()
```

Метод возвращает объект `Set` со всеми парами «ключ — значение» (в виде объектов `Map.Entry`) из этой карты.

`Map` имеет один субинтерфейс `SortedMap`. Он позволяет хранить пары «ключ — значение», отсортированные в конкретном порядке, в соответствии со значениями ключей. `SortedMap` предоставляет методы `subMap()`, `headMap()` и `tailMap()` для выборки подмножеств отсортированных карт. Как и `SortedSet`, он предоставляет метод `comparator()`, который возвращает объект, определяющий порядок сортировки ключей в карте. В Java 7 появился интерфейс `NavigableMap`, по своей функциональности сходный с `NavigableSet`: он добавляет методы для поиска ближайшего (по порядку сортировки) большего или меньшего значения по отношению к заданному.

Несмотря на очевидную взаимосвязь, `Map` формально не является типом `Collection` (`Map` не расширяет интерфейс `Collection`). Почему? Все методы интерфейса `Collection` вроде бы имеют смысл для `Map`, кроме `iterator()`. Однако `Map` содержит два набора объектов: ключи и значения, и для каждого требуется отдельный итератор. Вот почему `Map` не реализует `Collection`. Если вам нужно `Collection`-подобное представление `Map` с ключами и значениями, используйте метод `entrySet()`.

И еще одно замечание по поводу карт: некоторые реализации (включая стандартную коллекцию `Java HashMap`) позволяют использовать `null` в качестве ключей или значений, но другие такой возможности не поддерживают.

Ограничения типов

Суть обобщений — абстрагирование. Обобщения позволяют создавать классы и методы, работающие одинаково с объектами разных типов. Сам термин «обобщение» происходит от идеи написания обобщенных алгоритмов, которые можно было бы повторно использовать для многих типов объектов, вместо того чтобы адаптировать код к каждой конкретной ситуации. Эта концепция не нова; собственно, она лежит в основе всего объектно-ориентированного программирования. Обобщения Java не столько дополняют язык новыми возможностями, сколько упрощают написание и чтение повторно используемого кода.

Обобщения поднимают повторное использование на новый уровень: *тип* объектов, с которыми вы хотите работать, становится явным *параметром* обобщенного кода. По этой причине обобщения также называют *параметризованными типами*. Для обобщенного класса разработчик указывает тип как параметр (аргумент) при каждом использовании обобщенного типа. Класс параметризуется переданным типом, к которому адаптируется код.

В других языках обобщения иногда называются *шаблонами* (*templates*), но там речь скорее идет о реализации. Шаблоны похожи на промежуточные классы, которые ожидают получения параметров-типов, чтобы они могли использоваться в программе. В Java был выбран другой путь, у которого есть как свои достоинства, так и недостатки; они будут подробно рассмотрены в этой главе.

Об обобщениях Java можно рассказывать очень долго. Некоторые нюансы на первый взгляд могут показаться невразумительными, но не отчаивайтесь. Почти все, что вы будете делать с обобщениями, например использовать существующие классы вроде `List` и `Set`, — это очень просто и интуитивно понятно. Но для того, чтобы проектировать и писать ваши собственные обобщения, придется значительно лучше изучить тему и терпеливо экспериментировать.

Мы начнем рассматривать обобщения Java с самой очевидной сферы их применения — в классах контейнеров и в коллекциях, которые мы только что рассмотрели.

Затем мы сделаем шаг назад и рассмотрим сильные и слабые стороны обобщений. В завершение мы рассмотрим пару реальных обобщенных классов из Java API.

Контейнеры

В таких объектно-ориентированных языках, как Java, под *полиморфизмом* понимается то, что объекты всегда в какой-то степени заменяемы. Любой объект-потомок может использоваться вместо его родителя, поэтому каждый объект является потомком `java.lang.Object` — своего рода объектно-ориентированной «Евы». А значит, будет естественно, если самые универсальные типы *контейнеров* в Java будут работать с типом `Object`, чтобы в них можно было хранить практически любые объекты. Под контейнерами мы понимаем классы, способные хранить экземпляры других классов. Лучшим примером контейнеров служит API коллекций Java из предыдущего раздела. Напомним, что списки `List` содержат упорядоченные коллекции элементов типа `Object`. Множества `Map` содержат пары «ключ — значение», при этом ключи и значения относятся к самому общему типу `Object`. С небольшой поддержкой от оберток для примитивных типов такая схема работала достаточно хорошо. Тем не менее в каком-то смысле «коллекция объектов любых типов» также становится «коллекцией без типа», а работа с `Object` возлагает слишком большую ответственность на программиста.

Происходящее напоминает маскарад, на котором все объекты носят одинаковые маски и растворяются в толпе элементов коллекции. Когда объект «переодевается» в тип `Object`, компилятор перестает видеть его реальный тип. Впоследствии программист может нарушить анонимность объектов при помощи преобразования типов. И вам лучше проследить за тем, чтобы преобразование было правильным, иначе вы рискуете нарваться на неприятный сюрприз (как при попытке сдернуть накладную бороду с участника маскарада):

```
Date date = new Date();
List list = new ArrayList();
list.add( date );
...
Date firstElement = (Date)list.get(0); // Это правильное преобразование?
                                         // Может быть...
```

Интерфейс `List` также содержит метод `add()`, который получает любую разновидность `Object`. Здесь мы указали экземпляр `ArrayList`, который является реализацией интерфейса `List`, и добавили объект `Date`. Будет ли преобразование правильным в данном случае? Все зависит от того, что происходит в период времени, обозначенный «...». Конечно, компилятор Java знает, что подобные операции сопряжены с повышенным риском, и в настоящее время выдает предупреждения при добавлении элементов в простой массив `ArrayList`. В этом трудно убедиться при помощи `jshell`. После импортирования из пакетов `java`.

`util` и `javax.swing` попробуйте создать объект `ArrayList` и добавить несколько разнородных элементов:

```
jshell> import java.util.ArrayList;
jshell> import javax.swing.JLabel;
jshell> ArrayList things = new ArrayList();
things ==> []

jshell> things.add("Hi there");
| Warning:
| unchecked call to add(E) as a member of the raw type java.util.ArrayList
| things.add("Hi there");
| ^-----^
$3 ==> true

jshell> things.add(new JLabel("Hi there"));
| Warning:
| unchecked call to add(E) as a member of the raw type java.util.ArrayList
| things.add(new JLabel("Hi there"));
| ^-----^
$5 ==> true

jshell> things
things ==> [Hi there, javax.swing.JLabel[...text=Hi there,...]]
```

Предупреждения выдаются независимо от типа, указанного при вызове `add()`. На последнем шаге, при выводе содержимого `things`, в списке соседствуют объект `String` и объект `JLabel`. Компилятор не беспокоится об использовании разнородных типов; он просто любезно предупреждает вас, что ему неизвестно, будут ли преобразования типов вроде приведенного выше преобразования (`Date`) правильно работать на стадии выполнения.

Можно ли улучшить контейнеры?

Естественно спросить: нельзя ли как-то улучшить ситуацию? А если вам заранее известно, что в список должны включаться только объекты `Date`? Нельзя ли сделать собственный список, который принимает только объекты `Date`, избавиться от преобразования и снова пользоваться помощью компилятора? Как ни странно, нельзя (по крайней мере сколько-нибудь простым способом).

Первое естественное желание — попробовать «переопределить» методы `ArrayList` в субклассе. Но конечно, замена метода `add()` в субклассе ничего не переопределяет; просто в классе появляется новый *перегруженный* метод:

```
public void add( Object o ) { ... } // Все еще здесь
public void add( Date d ) { ... } // Перегруженный метод
```

Полученный объект будет принимать объекты любого типа — просто при этом будут вызываться разные методы.

Придется поискать другое решение. Например, можно написать собственный класс `DateList`, который не расширяет `ArrayList`, а делегирует реализацию своих методов реализации `ArrayList`. Прделав немалый объем однообразной работы, мы получим объект, который делает все то же, что делает `List`, но работает с объектами `Date` так, что компилятор и исполнительная среда знают их тип и не позволяют его нарушать. Но в результате этой работы мы окажем себе медвежью услугу, потому что наш контейнер перестанет быть реализацией `List` и мы не сможем использовать его со всеми средствами работы с коллекциями (такими, как `Collections.sort()`) или добавить его в другую коллекцию методом `addAll()` класса `Collection`.

Подведем итог: проблема заключается в том, что на самом деле мы хотим не уточнить поведение наших объектов, а изменить их «контракт с пользователем». Мы хотим адаптировать их API к более конкретному типу, а полиморфизм не позволяет это сделать. Похоже, обойтись без `Object` при работе с коллекциями не удастся? Но на помощь приходят обобщения.

Знакомство с обобщениями

Упомянутые в предыдущем разделе обобщения представляют собой расширение синтаксиса, позволяющее специализировать классы и методы для работы с конкретными типами. Обобщенный класс «настраивает себя» с помощью одного или нескольких *параметров-типов*, которые получает при каждом обращении к нему.

Например, заглянув в исходный код или в Javadoc-документацию класса `List`, вы найдете определение следующего вида:

```
public class List< E > {  
    ...  
    public void add( E элемент ) { ... }  
    public E get( int i ) { ... }  
}
```

Идентификатор `E` между угловыми скобками (`< >`) — это *параметр-тип* (*type parameter*)¹. Вся эта конструкция в угловых скобках показывает, что класс `List` является обобщенным и требует завершения (уточнения) путем передачи ему в аргументе какого-либо конкретного типа Java. Имя `E` выбрано произвольно,

¹ Другой вариант названия — переменная-тип (*type variable*). Мы предпочитаем термин «type parameter», который чаще используется в спецификации языка Java, но на практике вам могут встретиться оба варианта.

хотя есть соглашения, о которых мы скажем далее. В первой строке параметр-тип `E` обозначает тип элементов, которые должны храниться в списке. Далее класс `List` в своем теле и в своих методах обращается к `E` точно так же, как если бы вместо `E` был указан реальный тип (который будет подставлен на эти места позднее). Параметры-типы можно использовать в объявлениях переменных экземпляров, аргументов методов и возвращаемых типов методов. В данном случае `E` обозначает тип элементов, которые будут добавляться методом `add()`, а также возвращаемый тип метода `get()`. Давайте разберемся, как все это работает.

Чтобы использовать класс `List`, мы указываем нужный нам тип в таких же угловых скобках:

```
List<String> listOfStrings;
```

Здесь мы объявили переменную с именем `listOfStrings`, взяв за основу обобщенный тип `List` с параметром-типом `String`. Мы можем создать и другие специализированные версии `List` — с любыми другими типами классов Java. Примеры:

```
List<Date> dates;
```

```
List<java.math.BigDecimal> decimals;
```

```
List<Foo> foos;
```

Когда мы конкретизируем обобщенный тип таким образом, это называется *воплощением типа* (*instantiating the type*). Другие варианты названия — *инстанцирование типа*, *реализация типа*, *вызов типа*. Сравните: при работе с обычным типом Java мы просто обращаемся к нему по имени, а обобщенный тип надо воплощать при каждом использовании¹. Точнее говоря, тип должен быть воплощен повсюду, где он может появиться как объявляемый тип переменной (это показано в нашем примере), как тип аргумента метода, как возвращаемый тип метода, а также в командах создания объектов с ключевым словом `new`.

Вернемся к списку `listOfStrings`. Мы получили такой `List`, в котором тип `String` был подставлен компилятором вместо параметра-типа `E` в теле класса:

```
public class List< String > {
    ...
    public void add( String element ) { ... }
    public String get( int i ) { ... }
}
```

¹ Но кроме тех случаев, когда программист хочет использовать обобщенный тип обычным, «необобщенным» образом. Эти случаи мы скоро рассмотрим в разделе «Необработанные типы».

Мы специализировали класс `List` для работы только с элементами типа `String` — и ни с какими другими! Данная сигнатура методов уже не позволяет им принимать общий тип `Object`.

`List` — всего лишь интерфейс. Теперь, чтобы использовать полученную переменную, надо создать экземпляр некоторой реализации `List`. Для этого нам снова пригодится `ArrayList` как в рассмотренных ранее примерах. Как и прежде, `ArrayList` — это класс, реализующий интерфейс `List`, но в данном случае `List` и `ArrayList` являются обобщенными классами. Поэтому их надо инстанцировать при каждом использовании, указывая нужный параметр-тип. Мы, конечно, создадим такой `ArrayList`, который предназначен только для хранения элементов `String`, то есть соответствующий нашей переменной `listOfStrings`:

```
List<String> listOfStrings = new ArrayList<String>();  
// Или сокращенная запись для Java 7.0 и выше:  
List<String> listOfStrings = new ArrayList<>();
```

Как обычно, за ключевым словом `new` следует тип Java и круглые скобки с возможными аргументами для конструктора класса. В данном случае типом является `ArrayList<String>` — обобщенный тип `ArrayList`, воплощенный с типом `String`.

Объявлять переменные так, как показано в первой строке предыдущего примера, не совсем удобно, потому что параметр-тип приходится вводить дважды: в левой части и в инициализирующем выражении. В сложных случаях запись получается очень длинной: со многими параметрами-типами и даже со вложенными конструкциями. Начиная с Java 7, компилятору хватает сообразительности, чтобы определить тип инициализирующего выражения по типу той переменной, которой присваивается значение. Это называется *выведением типа*, или *автоматическим определением обобщенного типа* (*generic type inference*). Как показано в нашем примере, при сокращенной записи в правой части объявления переменной достаточно пустых угловых скобок: `< >`.

Теперь мы можем использовать свою специализированную версию `List` со строками. Компилятор пресекает любые попытки поместить в список что-то иное, кроме объектов типа `String` (или подтипа `String`, если бы он существовал), и дает возможность читать их методом `get()` без каких-либо преобразований:

```
jshell> ArrayList<String> listOfStrings = new ArrayList<>();  
listOfStrings ==> []  
  
jshell> listOfStrings.add("Hey!");  
$8 ==> true  
  
jshell> listOfStrings.add(new JLabel("Hey there"));  
| Error:  
| incompatible types: javax.swing.JLabel cannot be converted to java.lang.String
```

```
| listOfStrings.add(new JLabel("Hey there"));
|           ^-----^
jshell> String s = strings.get(0);
s ==> "Hey!"
```

Возьмем другой пример из API коллекций. Интерфейс `Map` реализует соответствия в стиле словарей: объекты-ключи ассоциируются с объектами-значениями. Ключи и значения не обязаны относиться к одному типу. Обобщенный интерфейс `Map` требует двух параметров-типов: для типа ключа и для типа значения. Javadoc-документация выглядит так:

```
public class Map< K, V > {
    ...
    public V put( K key, V value ) { ... } // Возвращает старое значение
    public V get( K key ) { ... }
}
```

Мы можем, например, создать `Map`, чтобы хранить объекты `Employee`, содержащие сведения о работниках предприятия, идентифицируемых целочисленными табельными номерами:

```
Map< Integer, Employee > employees = new HashMap< Integer, Employee >();
Integer bobsId = 314; // Спасибо автоупаковке!
Employee bob = new Employee( "Bob", ... );

employees.put( bobsId, bob );
Employee employee = employees.get( bobsId );
```

Здесь мы использовали `HashMap` — обобщенный класс, реализующий интерфейс `Map`, — и инстанцировали оба параметра-типа: `Integer` и `Employee`. Теперь `Map` работает только с ключами типа `Integer` и хранит значения типа `Employee`.

Причина, по которой для хранения числа использовалась обертка `Integer`, заключается в том, что параметры-типы обобщенных классов должны быть типами классов. Обобщенный класс нельзя параметризовать примитивным типом, например `int` или `boolean`. К счастью, благодаря автоупаковке примитивов в Java (см. «Обертки для примитивных типов», с. 174) мы можем использовать примитивные типы так же, как если бы они были типами-обертками.

Помимо API коллекций, десятки других API тоже используют обобщения, чтобы вы могли адаптировать их к конкретным типам. Мы поговорим о них в тех случаях, когда они будут встречаться в книге.

Несколько слов о типах

Прежде чем переходить к более важным вещам, надо сказать несколько слов о том, каким образом мы описываем конкретную параметризацию обобщенного

класса. Самый частый и очевидный способ применения обобщений — для работы с контейнерами, поэтому обобщенные типы часто воспринимаются как «средства для хранения» объектов, заданных параметрами-типами. В нашем примере мы назвали `List<String>` «списком строк», потому что по сути он именно этим и является. Аналогичным образом можно назвать нашу карту с информацией о работниках «картой, связывающей идентификаторы работников с объектами `Employee`». Тем не менее эти определения в большей степени сосредоточены на том, что **делают** классы, чем на самих типах. Но можно взглянуть на обобщенные типы иначе: для примера возьмем контейнер с единственным объектом, имеющий имя `Trap<E>`, и создадим его экземпляры для объектов типа `Mouse` и типа `Bear`, то есть `Trap<Mouse>` или `Trap<Bear>`. Будет вполне естественно называть эти новые типы «мышинная ловушка» и «медвежья ловушка». Точно так же можно воспринимать `List<String>` как новый тип «строковый список». А нашу карту сотрудников можно воспринимать как новый тип «целочисленная карта объектов сотрудников». Выбирайте тот вариант формулировки, который больше нравится, но обратите внимание, что во втором варианте обобщение выглядит как **тип**. Это может помочь вам сохранить ясность терминов, когда мы будем обсуждать, как обобщенные типы связаны в системе типов. В таком контексте «контейнерное» восприятие окажется не совсем логичным.

В следующем разделе мы продолжим обсуждение обобщенных типов Java с другой точки зрения. Вы уже видели, **что** они могут сделать; пора разобраться, **как** они это делают.

«Ложки не существует»

В фильме «Матрица»¹ Нео оказывается перед выбором. Взять синюю таблетку и остаться в вымышленном мире? Или взять красную таблетку и увидеть все в истинном свете? Работая с обобщениями в Java, мы сталкиваемся с похожей онтологической дилеммой. Рано или поздно при рассмотрении обобщений нам придется столкнуться с вопросом об их практической реализации в компьютере. Наш вымышленный мир создается компилятором, чтобы нам было проще писать код. Реальность (пусть и не мрачная антиутопия, показанная в фильме) — это

¹ Для тех, кому непонятно название раздела, приведем цитату:

Мальчик. Не пытайся согнуть ложку. Это невозможно. Сначала надо понять главное.

Нео. Что главное?

Мальчик. Ложки не существует.

Нео. Не существует?

Мальчик. Знаешь, это не ложка гнется. Все — обман. Дело в тебе».

(Братья Вачовски, «Матрица». 136 минут. Warner Brothers, 1999.)

более суровое место, полное скрытых опасностей и неудобных вопросов. Почему преобразования типов не могут правильно работать с обобщениями? Почему я не могу реализовать что-то вроде двух разных обобщенных интерфейсов в одном классе? Почему я могу объявить массив обобщенных типов, хотя в Java нельзя создавать такие массивы? В этой главе мы ответим на эти и другие вопросы, и вам даже не придется ждать продолжения фильма. Скоро вы научитесь «сгибать ложки» (вернее, типы).

Проектируя обобщения Java, разработчики решали сложнейшую задачу: добавить в популярный язык совершенно новый синтаксис, который безопасно введет параметризованные типы без ущерба для быстродействия... а заодно сохранить обратную совместимость со всем существующим кодом Java и избежать значительных изменений в уже скомпилированных классах. Удивительно, что разработчики вообще справились с этой задачей. Разумеется, им понадобилось немало времени. И при этом, как часто бывает, пришлось идти на компромиссы, что породило ряд проблем.

Стирание типов

Для решения этой задачи в Java используется прием, называемый *стиранием типов* (*erasure*). Практически все, что мы делаем с обобщениями, происходит статически на стадии компиляции. Поэтому информацию об обобщениях можно не включать в скомпилированные классы! Обобщенные свойства классов, предоставленные компилятором, стираются (удаляются) в скомпилированных классах — так достигается совместимость с обычным, «необобщенным» кодом. Хотя Java сохраняет информацию об обобщенных свойствах классов в скомпилированной форме, эта информация используется главным образом компилятором. А исполнительная среда Java ничего не знает об обобщениях.

Рассмотрим скомпилированный обобщенный класс: уже знакомый нам список `List`. Для этого можно воспользоваться командой `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 точно так же, как он выглядел до внедрения в язык обобщений (в этом можно убедиться при помощи любой старой версии JDK). Обратите внимание, что методы `add()` и `get()` используют тип элементов `Object`. Вы думаете, что это какая-то хитрость и при воплощении типа где-то в глубине Java появится новая версия класса? Нет, это не так. Существует только один

класс `List`, и именно он является реальным типом для всех параметризаций `List`, например для `List<Date>` и `List<String>`, в чем нетрудно убедиться:

```
List<Date> datelist = new ArrayList<Date>();
System.out.println( datelist instanceof List ); // true!
```

Но наш обобщенный `datelist` не реализует только что упомянутые методы класса `List`:

```
datelist.add( new Object() ); // Ошибка компиляции
```

В этом проявляется слегка шизофреническая природа обобщений Java. Компилятор в них верит, но исполнительная среда знает, что они — всего лишь иллюзия. Попробуем трезво взглянуть на эту ситуацию и проверить, что `datelist` относится к типу `List<Date>`:

```
System.out.println( datelist instanceof List<Date> ); // Ошибка компиляции!
// Недопустимо: обобщенный тип для instanceof
```

На этот раз компилятор просто опускает шлагбаум и говорит: «Нет». Обобщенный тип нельзя проверить операцией `instanceof`. Во время выполнения не существует фактически отличающихся классов для разных параметризаций `List`, поэтому оператор `instanceof` не может отличить один `List` от другого. Вся проверка безопасности обобщений осталась на стадии компиляции, и теперь мы просто работаем с одним реальным типом `List`.

В действительности произошло следующее: компилятор стер все синтаксические элементы с угловыми скобками и заменил параметры-типы в нашем классе `List` таким типом, который может работать во время выполнения с любым допустимым типом, в данном случае это `Object`. Похоже, мы вернулись к исходной точке, не считая того, что у компилятора теперь есть информация, позволяющая ему проверять все наши действия с обобщениями на стадии компиляции. Следовательно, он может выполнять преобразования типов вместо нас. Попробуйте декомпилировать класс `List<Date>` (команда `javap` с ключом `-c` выводит байт-код, если вы не из пугливых), и вы увидите, что в скомпилированном коде есть преобразование в `Date`, хотя в исходном коде его не было.

Теперь можно ответить на один из вопросов, поставленных в начале раздела: «Почему я не могу реализовать что-то вроде двух разных обобщенных интерфейсов в одном классе?» Класс не может реализовать два разных воплощения обобщенного типа `List`, потому что во время выполнения они становятся одним типом и различить их невозможно:

```
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>
```

К счастью, всегда есть обходные решения. Например, в данном случае можно использовать общий суперкласс или создать несколько классов. Возможно, альтернативы не будут блистать элегантностью, но почти всегда удастся найти корректное решение, даже если оно оказывается длиннее, чем вам хотелось бы.

Необработанные типы

Хотя во время компиляции разные параметризации обобщенного типа рассматриваются компилятором как разные типы (с разными API), вы уже знаете, что во время выполнения существует только один реальный тип. Например, классы `List<Date>` и `List<String>` совместно используют старый класс Java `List`. Его называют *необработанным типом* (*raw type*), или *сырым типом*, обобщенного класса. У каждого обобщенного типа есть свой необработанный тип. Это вырожденная, «тривиальная» для Java форма, из которой удалена вся информация об обобщенных типах, то есть вместо параметров-типов используется общий тип Java (такой, как `Object`)¹.

Вы можете работать с необработанными типами точно так же, как это было до включения обобщений в язык. Единственное отличие заключается в том, что компилятор выдает предупреждение при их «небезопасном» использовании. За пределами `jshell` компилятор умеет распознавать такие проблемы:

```
// Необобщенный код Java, использующий необработанный тип
List list = new ArrayList(); // Присваивание работает
list.add("foo"); // Предупреждение компилятора об использовании
                  // необработанного типа
```

В этом фрагменте необработанный тип `List` используется «в старом стиле» (как было принято до выхода Java 5). Поэтому при добавлении объекта в список компилятор предупреждает, что эта операция непроверяемая или небезопасная:

```
$ javac MyClass.java
Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Компилятор предлагает использовать ключ `-Xlint:unchecked` для получения более конкретной информации о местонахождении небезопасных операций:

¹ Когда в Java 5.0 появились обобщения, разработчики языка тщательно позаботились о том, чтобы необработанный тип каждого обобщенного класса ничем не отличался от более раннего, необобщенного типа. Таким образом, необработанный тип `List` в Java 5.0 не отличается от старого, необобщенного типа `List`, который появился еще в JDK 1.2. Поскольку практически весь код Java, существовавший на тот момент, не содержал обобщений, эквивалентность и совместимость типов была очень важна.

```
$ 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");
```

Учтите, что при создании и присваивании необработанного типа `ArrayList` предупреждение не выдается. Оно будет получено только при вызове «небезопасного» метода (содержащего обращение к параметру-типу). Это означает, что вы можете использовать старые, «необобщенные» Java API, работающие с необработанными типами. Предупреждения будут выдаваться только при попытке выполнения небезопасных операций в вашем коде.

И еще несколько слов о стирании типов, прежде чем двигаться дальше. В предыдущих примерах параметры-типы заменялись типом `Object`, который может представлять любой тип, применимый к параметру-типу `E`. Сейчас вы увидите, что это не всегда так. Для параметров-типов можно установить ограничения, и при этом компилятор отнесется к стиранию типов более избирательно:

```
class Bounded< E extends Date > {  
    public void addElement( E element ) { ... }  
}
```

Такое объявление параметра-типа означает, что тип элемента `E` обязательно должен быть подтипом типа `Date`. Поэтому при стирании типа в методе `addElement()` компилятор не дойдет до уровня `Object`, а остановится на `Date`:

```
public void addElement( Date element ) { ... }
```

Здесь `Date` является *верхней границей* (*upper bound*) указанного типа, то есть `Date` в данном случае находится на вершине иерархии объектов. Значит, воплощение типа возможно только с `Date` или «нижестоящими» (производными от него) типами.

Итак, теперь вы примерно представляете, как работают обобщенные типы, и мы можем рассмотреть их поведение чуть подробнее.

Отношения между параметризованными типами

Мы знаем, что параметризованные типы совместно используют единый необработанный тип. Именно из-за этого наш параметризованный `List<Date>` во время выполнения представляет собой обычный `List`. Таким образом, переменной необработанного типа можно присвоить любое воплощение `List`:

```
List list = new ArrayList<Date>();
```

А можно даже пойти в другом направлении и присвоить необработанный тип конкретному воплощению обобщенного типа:

```
List<Date> dates = new ArrayList(); // Предупреждение
```

Эта команда генерирует предупреждение о непроверяемой операции при попытке присваивания, но в дальнейшем компилятор полагает, что до присваивания список содержал только объекты `Date`. Также допустимо, хотя и бессмысленно, выполнить в этой команде преобразование типа. О преобразовании к обобщенным типам мы вкратце расскажем в разделе «Преобразования типов», с. 256.

Какими бы ни были типы на стадии выполнения, компилятор «командует парадом» и не позволяет присваивать явно несовместимые объекты:

```
List<Date> dates = new ArrayList<String>(); // Ошибка компиляции!
```

Конечно, `ArrayList<String>` не реализует методы типа `List<Date>`, сформированного компилятором, поэтому эти типы несовместимы.

Но как насчет более интересных отношений между типами? Например, интерфейс `List` является подтипом более общего интерфейса `Collection`. Будет ли конкретное воплощение обобщения `List` совместимо по присваиванию с каким-либо воплощением обобщенного типа `Collection`? Зависит ли это от параметров-типов и отношений между ними? Очевидно, `List<Date>` не является типом `Collection<String>`. Но является ли `List<Date>` типом `Collection<Date>`? Может ли `List<Date>` быть типом `Collection<Object>`?

Сначала мы просто ответим, а потом займемся анализом и пояснениями. Правило гласит, что для простых видов воплощений обобщенных типов, которые рассматривались до сих пор, наследование применяется только к «базовому» обобщенному типу, но не к параметрам-типам. Более того, совместимость по присваиванию применима только в том случае, когда два обобщенных типа воплощаются одним и тем же параметром-типом. Иначе говоря, все еще существует одномерное наследование от базового типа обобщенного класса, но с дополнительным ограничением, согласно которому параметры-типы должны быть идентичны.

Например, так как `List` является типом `Collection`, мы можем присваивать воплощения `List` воплощениям `Collection` при точном совпадении параметра-типа:

```
Collection<Date> cd;  
List<Date> ld = new ArrayList<Date>();  
cd = ld; // ОК!
```

Из этого фрагмента следует, что `List<Date>` является типом `Collection<Date>` — вполне логично. Но попытка применить ту же логику при расхождении параметров-типов приводит к неудаче:

```
List<Object> lo;  
List<Date> ld = new ArrayList<Date>();  
lo = ld; // Ошибка компиляции! Несовместимые типы
```

Хотя интуиция подсказывает, что объекты `Date` в `List` могли бы прекрасно существовать как объекты `Object` в `List`, попытка присваивания приводит к ошибке. В следующем разделе мы подробно объясним, почему это происходит, а пока заметим, что параметры-типы не совпадают и между параметрами-типами в обобщениях не существует отношений наследования. В данном случае лучше думать о воплощениях в контексте типов, а не в контексте того, что они делают. Мы имеем дело не со «списком дат» и «списком объектов», а скорее с `DateList` и `ObjectList`, отношения между которыми не столь очевидны.

Следующий пример поможет понять, что можно, а чего нельзя делать в таких ситуациях:

```
Collection<Number> cn;  
List<Integer> li = new ArrayList<Integer>();  
cn = li; // Ошибка компиляции! Несовместимые типы
```

Воплощение `List` может быть воплощением `Collection`, но только в том случае, если их параметры-типы в точности совпадают. Наследование не распространяется на параметры-типы, и в этом примере возникает ошибка.

Мы упоминали, что это правило применяется к уже рассмотренным простым видам воплощений. Какие еще виды существуют? Те воплощения, которые рассматривались до настоящего момента, с передачей реального типа Java в качестве параметра, называются *конкретными воплощениями типа* (*concrete type instantiations*). Далее мы упомянем и об *универсальных воплощениях* (*wildcard instantiations*), сходных с математическими операциями над множествами. Вы увидите, что возможны и более экзотические виды обобщений с двумерными отношениями между типами, с зависимостями как от базового типа, так и от параметризации. Не беспокойтесь: они встречаются не очень часто и не так страшны, как кажутся на первый взгляд.

Почему `List<Date>` не является `List<Object>`?

Вполне логичный вопрос. Даже если думать о `DateList` и `ObjectList` как о разных типах, все равно можно спросить, не совместимы ли они по присваиванию. Почему нельзя иметь возможность присвоить `List<Date>` переменной `List<Object>` и работать с элементами `Date` как с типами `Object`?

Дело в главной причине для применения обобщений, которая упоминалась во вводном разделе: изменении API. В простейшем случае, если предположить, что тип `ObjectList` расширяет тип `DateList`, последний будет содержать все мето-

ды `ObjectList` и в него можно будет вставлять элементы `Object`. На это можно возразить, что обобщения позволяют нам изменять API, так что этот довод не действует. Это верно, но есть и более серьезная проблема. Если бы мы могли присвоить `DateList` переменной `ObjectList`, то мы могли бы использовать методы `Object` для вставки в него элементов типов, отличных от `Date`. Тогда можно было бы определить для `DateList` синоним (то есть предоставить альтернативный, более широкий тип) в форме `ObjectList` и попытаться обмануть его, заставив принять другой тип:

```
DateList dateList = new DateList();
ObjectList objectList = dateList; // Так нельзя
objectList.add( new Foo() ); // Должна произойти ошибка времени выполнения!
```

При попытке использования фактической реализации `DateList` с неправильным типом объекта должна происходить ошибка времени выполнения. И здесь скрывается проблема: обобщения Java не имеют своего представления на стадии выполнения. Даже если бы эта функциональность была полезной, в текущей схеме реализации Java невозможно определить, что в связи с этим нужно делать во время выполнения. На проблему можно взглянуть иначе: такая возможность была бы попросту опасной, потому что она допускает ошибки времени выполнения, которые невозможно перехватить во время компиляции. Но мы, конечно, хотим, чтобы ошибки типов всегда выявлялись во время компиляции.

Можно подумать, что запретом таких присваиваний Java гарантирует безопасность типов вашего кода, если он компилируется без предупреждений. К сожалению, таких гарантий нет, но проблема связана не с обобщениями, а с массивами. Если это замечание покажется вам знакомым, так это потому, что мы уже упоминали об этом — применительно к массивам Java. Типы массивов обладают отношением наследования, которое допускает такую синонимию:

```
Date [] dates = new Date[10];
Object [] objects = dates;
objects[0] = "not a date"; // Исключение ArrayStoreException!
```

Однако массивы на стадии выполнения представляются другими классами, и они проверяют себя во время выполнения, выдавая исключение `ArrayStoreException` в подобных ситуациях. Это значит, что при таком использовании массивов компилятор теоретически не может гарантировать безопасность типов в коде Java.

Преобразования типов

Мы обсудили отношения между обобщенными типами — и даже между обобщенными и необработанными типами. Однако мы еще не исследовали концепцию преобразований типов в мире обобщений. При совмещении обобщений с необ-

работанными типами никакие преобразования не были нужны. Вместо этого мы просто пересекали линию, на которой компилятор выдавал предупреждения о непроверяемых операциях:

```
List list = new ArrayList<Date>();  
List<Date> dl = list; // Предупреждение
```

В обычной ситуации преобразование типа в Java используется для работы с двумя типами, которые могут быть совместимыми по присваиванию. Например, можно попытаться преобразовать `Object` в `Date`, потому что `Object` может быть значением `Date`. Тогда операция преобразования выполняет проверку на правильность во время выполнения. Преобразование между несвязанными типами вызывает ошибку компиляции. Например, даже не стоит пытаться преобразовать `Integer` в `String`. Такие типы не связаны отношениями наследования. А как насчет преобразований совместимых обобщенных типов?

```
Collection<Date> cd = new ArrayList<Date>();  
List<Date> ld = (List<Date>)cd; // OK!
```

В этом фрагменте продемонстрировано допустимое преобразование более общего типа `Collection<Date>` в `List<Date>`. Такое преобразование возможно, потому что тип `Collection<Date>` совместим по присваиванию и может содержать `List<Date>`. Аналогичным образом следующее преобразование перехватывает нашу ошибку, когда мы определяем для `TreeSet<Date>` синоним `Collection<Date>` и пытаемся преобразовать его в `List<Date>`:

```
Collection<Date> cd = new TreeSet<Date>();  
List<Date> ld = (List<Date>)cd; // Исключение ClassCastException!  
ld.add( new Date() );
```

Тем не менее есть случай, когда преобразования типов неэффективны с обобщениями, а именно при попытке различить типы по их параметрам-типам:

```
Object o = new ArrayList<String>();  
List<Date> ld = (List<Date>)o; // Предупреждение о непроверяемой операции,  
                             // она бесполезна  
Date d = ld.get(0); // Небезопасно во время выполнения,  
                   // неявное преобразование может завершиться неудачей
```

Здесь мы определяем для `ArrayList<String>` синоним в виде простого типа `Object`, который затем преобразуется в `List<Date>`. К сожалению, Java не может отличить `List<String>` от `List<Date>` во время выполнения, поэтому преобразование бесполезно. Компилятор сообщает об этом, выдавая предупреждение о непроверяемой операции в точке преобразования. Вы должны это учитывать, когда будете использовать преобразованный объект впоследствии — он может оказаться неправильным. Преобразования с обобщенными типами неэффек-

тивны во время выполнения из-за стирания типов, то есть из-за отсутствия информации о типах.

Преобразования между коллекциями и массивами

Преобразования между коллекциями и массивами выполняются просто. Для удобства элементы коллекции можно получить в виде массива при помощи следующих методов:

```
public Object[] toArray()  
public <E> E[] toArray( E[] a )
```

Первый метод возвращает простой массив `Object`. Во второй форме можно действовать более конкретно и получить массив с элементами правильного типа. Если предоставить массив достаточного размера, он будет заполнен значениями. Но если массив окажется слишком коротким (например, нулевой длины), будет создан и возвращен новый массив **того же типа, но с необходимой длиной**. Таким образом, вы можете просто передать пустой массив с правильным типом:

```
Collection<String> myCollection = ...;  
String [] myStrings = myCollection.toArray( new String[0] );
```

(Этот трюк выглядит немного неуклюже, и было бы лучше, если бы язык Java позволял явно задать тип ссылкой `Class`, но по какой-то причине это невозможно.) Также можно пойти в обратную сторону и преобразовать массив объектов в коллекцию `List` статическим методом `asList()` класса `java.util.Arrays`:

```
String [] myStrings = ...;  
List list = Arrays.asList( myStrings );
```

Итератор

Итератор — это объект для перебора последовательности значений. Эта операция встречается так часто, что для нее был определен стандартный интерфейс: `java.util.Iterator`. Интерфейс `Iterator` имеет всего два основных метода:

```
public E next()
```

Метод возвращает следующий элемент (элемент обобщенного типа `E`) перебираемой коллекции.

```
public boolean hasNext()
```

Метод возвращает `true`, если вы еще не перебрали все элементы `Collection`. Другими словами, он возвращает `true`, если возможно вызвать `next()` для получения следующего элемента.

Этот пример демонстрирует использование `Iterator` для вывода каждого элемента коллекции:

```
public void printElements(Collection c, PrintStream out) {
    Iterator iterator = c.iterator();
    while ( iterator.hasNext() ) {
        out.println( iterator.next() );
    }
}
```

Кроме методов перебора, `Iterator` предоставляет возможность удаления элементов из коллекции:

```
public void remove()
```

Метод удаляет последний объект, возвращенный вызовом `next()`, из перебираемой коллекции.

Не все итераторы реализуют `remove()`. Например, возможность удаления элемента из коллекции, доступной только для чтения, не имеет смысла. Если удаление элемента запрещено, метод выдает исключение `UnsupportedOperationException`. Если `remove()` вызывается до первого вызова `next()` или если `remove()` вызывается дважды подряд, выдается исключение `IllegalStateException`.

Цикл `for` с коллекциями

Разновидность цикла `for`, описанная в разделе «Цикл `for`», с. 134, может работать со всеми типами `Iterable`; это означает, что она может использоваться для перебора любых объектов `Collection`, так как этот интерфейс расширяет `Iterable`. Например, можно перебрать все элементы типизованной коллекции объектов `Date`:

```
Collection<Date> col = ...
for( Date date : col )
    System.out.println( date );
```

Эта разновидность встроенных циклов `for` в языке Java называется расширенным циклом `for` (в отличие от числовых циклов `for`, которые появились задолго до обобщений). Расширенный цикл `for` применим только к коллекциям типа `Collection`, но не к `Map`. Контейнер `Map` — совсем другое дело. Он содержит два разных набора объектов (ключи и значения), поэтому неочевидно, что именно должен делать цикл. Но поскольку идея перебора карты в цикле выглядит разумно, вы можете воспользоваться двумя методами `Map`: `keySet()` или `values()` (или даже `entrySet()`, если вы действительно хотите, чтобы каждая пара «ключ — значение» представлялась отдельной сущностью), чтобы получить из вашей карты подходящую коллекцию, которая **будет** работать с расширенным циклом `for`.

Метод sort()

Изучая класс `java.util.Collections`, мы находим в нем разнообразные статические методы для работы с коллекциями. Среди них есть очень интересный представитель — статический обобщенный метод `sort()`:

```
<T extends Comparable<? super T>> void sort( List<T> list ) { ... }
```

Похоже, очередная головоломка. Давайте сосредоточимся на этом фрагменте:

```
Comparable<? super T>
```

Это универсальное воплощение интерфейса `Comparable`, поэтому `extends` можно читать как `implements`, если вам так будет удобнее. `Comparable` содержит метод `compareTo()` для некоторого параметра-типа. `Comparable<String>` означает, что метод `compareTo()` получает тип `String`. Следовательно, `Comparable<? super T>` — это набор воплощений `Comparable` для `T` и всех его суперклассов. Подойдет не только `Comparable<T>`, но и воплощение на другом конце иерархии: `Comparable<Object>`. Проще говоря, для метода `sort()` элементы списка должны быть сравнимы с другими элементами своих типов или любых супер-типов. Этого достаточно для полной уверенности в том, что все элементы могут сравниваться друг с другом, и это не так сильно вас ограничивает, как если бы вам пришлось использовать для всех элементов метод `compareTo()`. Некоторые элементы могут наследовать интерфейс `Comparable` от родительского класса, который умеет выполнять сравнение только с супертипом `T`, — это именно то, что разрешено в данном случае.

Приложение: деревья на поле

В этой главе содержится много теории. Не бойтесь теории — она поможет вам прогнозировать логику работы в незнакомых ситуациях и вдохновит на поиск решений новых задач. Но практика не менее важна, поэтому мы попробуем применить некоторые из этих коллекций на практике. Для этого мы вернемся к примеру игры, работа над которым началась в разделе «Классы», с. 68. В частности, пришло время разобраться с хранением нескольких объектов каждого типа.

В главе 11 будут рассмотрены сетевые коммуникации, а также предложена конфигурация для нескольких игроков, в которой потребуются хранить несколько физиков. Пока в игре поддерживается только один физик, который может кидать одно яблоко за раз. Но поле можно заполнить несколькими деревьями для тренировки. Ньютон еще покажет себя!

Добавим шесть деревьев, хотя мы будем использовать пару циклов, чтобы количество деревьев можно было легко увеличить. Сейчас в `Field` хранится один

экземпляр дерева. Мы можем преобразовать `Field` в типизованный список, после чего можно будет добавлять и удалять деревья несколькими способами. Мы можем создать для `Field` несколько методов, которые работают со списком и, возможно, обеспечивают соблюдение других правил игры (например, максимальное количество деревьев). С таким же успехом можно использовать список напрямую, так как класс `List` уже содержит удобные методы для большинства операций, которые мы собираемся выполнять. А можно воспользоваться сочетанием этих подходов — специальные методы там, где это имеет смысл, и прямые операции в остальных местах.

Так как у нас есть игровые правила, относящиеся к `Field`, мы воспользуемся первым вариантом. (Просмотрите приведенные примеры и подумайте, как бы вы изменили их для прямой работы со списком деревьев.) Начнем с метода `addTree()`. Одно из преимуществ выбранного подхода заключается в том, что создание экземпляра дерева можно переместить в наш метод, вместо того чтобы напрямую создавать экземпляр и работать с ним. Один из вариантов добавления дерева в заданной точке поля выглядит так:

```
public void addTree(int x, int y) {
    Tree tree = new Tree();
    tree.setPosition(x,y);
    trees.add(tree);
}
```

Этим методом мы легко добавим пару деревьев:

```
Field field = new Field();
...
field.addTree(100,100);
field.addTree(200,100);
```

Эти две строки добавляют пару деревьев, расположенных рядом друг с другом. Сделаем следующий шаг и напишем циклы, необходимые для создания шести деревьев:

```
Field field = new Field();
...
for (int row = 1; row <= 2; row++) {
    for (int col = 1; col <=3; col++) {
        field.addTree(col * 100, row * 100);
    }
}
```

Вероятно, вы поняли, как легко можно добавить восемь, девять или сто деревьев. Как мы уже отмечали, компьютеры отлично подходят для повторяющихся задач.

Ура, лес мишеней создан! Впрочем, мы пока пропустили ряд критически важных подробностей, и самая важная из них — вывод леса на экран. Необходимо

обновить метод прорисовки для класса `Field` так, чтобы он понимал и правильно использовал список деревьев. Затем то же самое будет сделано для физиков и яблок, когда мы займемся добавлением новой функциональности в игру. Также потребуется возможность удаления неактивных элементов. Но сначала — лес!

```
protected void paintComponent(Graphics g) {  
    g.setColor(fieldColor);  
    g.fillRect(0,0, getWidth(), getHeight());  
    for (Tree t : trees) {  
        t.draw(g);  
    }  
    physicist.draw(g);  
    apple.draw(g);  
}
```

Так как мы уже находимся в классе `Field`, в котором хранятся деревья, нет необходимости писать отдельную функцию для чтения и прорисовки каждого дерева. Можно воспользоваться удобной альтернативной структурой цикла `for` и быстро извлечь все деревья из списка. Результат показан на рис. 7.1. Красота!

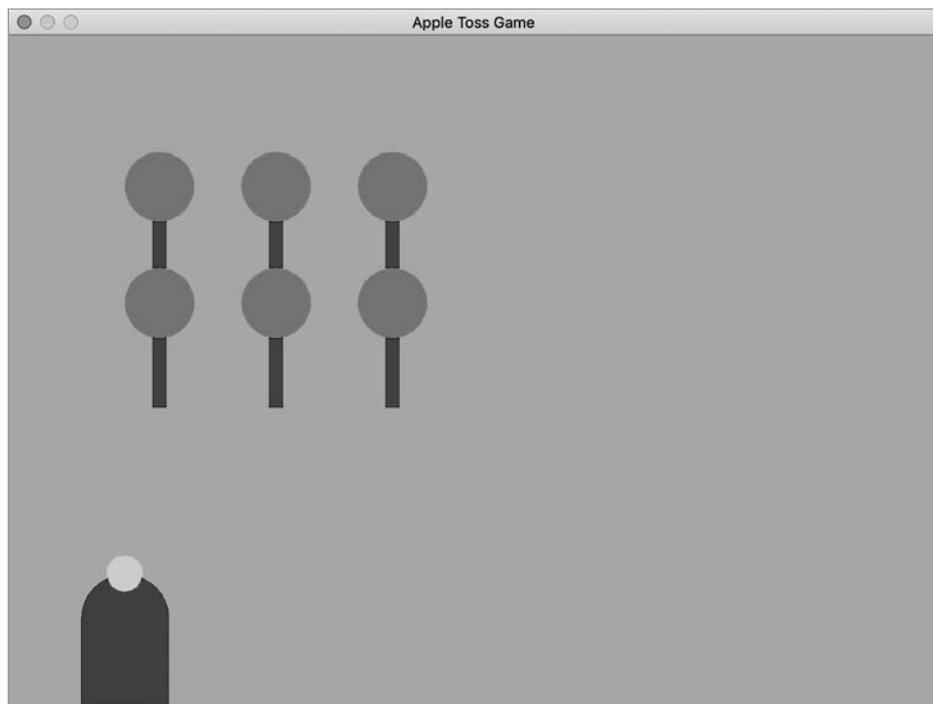


Рис. 7.1. Прорисовка всех деревьев из `List`

Заключение

Коллекции и обобщения — чрезвычайно мощные и полезные дополнения языка Java. И хотя некоторые подробности, в которые мы погрузились во второй половине этой главы, выглядят устрашающе, основной аргумент в пользу обобщений прост и неоспорим: они улучшают коллекции. Когда вы начнете писать больше кода, в котором используются обобщения, вы увидите, что ваш код лучше читается и становится более понятным. Коллекции обеспечивают элегантный, эффективный механизм хранения данных. Обобщения явно выражают то, о чем раньше приходилось догадываться по контексту.

ГЛАВА 8

Текст, числа, дата и время

Если вы читаете эту книгу по порядку, то уже знаете основные концепции Java, включая объектно-ориентированные аспекты языка и многопоточность. Пришло время переключиться на интерфейс программирования приложений Java, или API (Application Programming Interface), — набор классов, составляющих стандартные пакеты Java и входящих в каждую реализацию Java. Фундаментальные пакеты Java являются одной из самых выдающихся особенностей языка. Многие другие объектно-ориентированные языки обладают похожей функциональностью, но ни в одном из них нет такого набора стандартизированных API и инструментов, как в Java. Это и результат, и причина успеха Java.

Строки

Начнем с более глубокого рассмотрения класса Java `String` (или более конкретно, `java.lang.String`). Так как операции с объектами `String` считаются фундаментальными, важно понять, как они реализованы и что с ними можно делать. Объект `String` инкапсулирует последовательность символов Unicode. Во внутреннем представлении эти символы хранятся в обычном массиве Java, но объект `String` ревностно защищает массив и предоставляет доступ к нему только через свой собственный API. Это означает, что объекты `String` являются *неизменяемыми* (*immutable*); после того, как вы создадите объект `String`, изменить его значение уже не удастся. Вам может показаться, что многие операции с объектами `String` меняют символы или длину строки, но на самом деле они возвращают новый объект `String`, который копирует оригинал или содержит внутренние ссылки на необходимые символы оригинала. В некоторых реализациях Java компилятор старается объединить найденные в одном классе идентичные строки в пул общих строк и совместно использовать его фрагменты в разных местах байт-кода, если это возможно.

Изначально такое решение было принято ради быстродействия. Неизменяемые объекты `String` экономят память и могут оптимизироваться для скорости виртуальной машины Java. С другой стороны, программист должен хотя бы в общих чертах знать, как работает класс `String`, чтобы избежать создания лишних

объектов `String` там, где критично быстрое действие. Это было особенно важно в прошлом, когда виртуальные машины были медленными и неэффективно работали с памятью. В наши дни работа со строками обычно не создает проблем на фоне общей производительности реального приложения¹.

Создание строк

Строковые литералы, определяемые в вашем исходном коде, заключаются в двойные кавычки и могут присваиваться переменным `String`:

```
String quote = "To be or not to be";
```

Java автоматически преобразует строковый литерал в объект `String` и присваивает его переменной.

Объекты `String` знают свою длину, так что объектам `String` в Java не требуются специальные завершители. Для получения длины `String` можно воспользоваться методом `length()`. Также можно проверить, имеет ли строка нулевую длину, при помощи метода `isEmpty()`:

```
int length = quote.length();
boolean empty = quote.isEmpty();
```

К объектам `String` можно применять оператор `+` (единственный перегруженный оператор Java) для выполнения конкатенации. Следующие две команды создают эквивалентные строки:

```
String name = "John " + "Smith";
String name = "John ".concat("Smith");
```

Строковые литералы не могут (пока не могут²) занимать несколько строк в исходных файлах Java, но вы можете выполнить конкатенацию строк для достижения того же эффекта:

```
String poem =
    "'Twas brillig, and the slithy toves\n" +
    "    Did gyre and gimble in the wabe:\n" +
    "All mimsy were the borogoves,\n" +
    "    And the mome raths outgrabe.\n";
```

¹ Не верите — проверьте! Если ваш код, работающий со `String`, прост и понятен, не переписывайте его, пока кто-то не докажет вам, что он работает слишком медленно. Вполне возможно, что ваш критик ошибается. И пусть вас не смущают относительные сравнения. Миллисекунда в 1000 раз больше микросекунды, но и она может оказаться пренебрежимо малой для общего быстрого действия вашего приложения.

² В Java 13 реализована предварительная версия многострочных строковых литералов: <https://openjdk.java.net/jeps/355>.

Впрочем, лучше не встраивать длинный текст в исходный код. В главе 11 мы поговорим о способах загрузки строк из файлов и веб-источников. Помимо создания строк из литеральных выражений, вы можете создать строку непосредственно из массива символов:

```
char [] data = new char [] { 'L', 'e', 'm', 'm', 'i', 'n', 'g' };
String lemming = new String( data );
```

Также можно составить строку 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` содержит имя кодировки, то есть схемы кодирования символов. Конструктор `String` использует его для преобразования низкоуровневых байтов заданной кодировки в используемую во внутренней реализации кодировку, выбранную исполнителем среды. Если кодировка не указана, то используется кодировка по умолчанию для вашей системы¹.

И наоборот, метод `charAt()` класса `String` позволяет обращаться к символам `String` в синтаксисе, сходном с синтаксисом массива:

```
String s = "Newton";
for ( int i = 0; i < s.length(); i++ )
    System.out.println( s.charAt( i ) );
```

Этот цикл выводит содержимое строки по одному символу.

Представление о том, что `String` является последовательностью символов, также отражено тем фактом, что класс `String` реализует интерфейс `java.lang.CharSequence`. Этот интерфейс предписывает наличие методов `length()` и `charAt()` для получения подмножеств символов.

Создание строк по другим данным

Объекты и примитивные типы в Java могут быть преобразованы в текстовое представление в формате `String`. Для примитивных типов (например, числовых) строка формируется достаточно очевидно; для объектных типов все определяется самим объектом. Строковое представление значения можно получить при помощи статического метода `String.valueOf()`. Различные перегруженные версии этого метода получают каждый из примитивных типов:

¹ На большинстве платформ по умолчанию используется кодировка UTF-8. За подробной информацией о кодировках, настройках по умолчанию и стандартных кодировках, используемых Java, обращайтесь к официальной Javadoc-документации класса `java.nio.charset.Charset`.

```
String one = String.valueOf( 1 ); // Целое число, "1"
String two = String.valueOf( 2.384f ); // Число с плавающей точкой, "2.384"
String notTrue = String.valueOf( false ); // Логическое значение, "false"
```

У всех объектов в Java есть метод `toString()`, унаследованный от класса `Object`. Для многих объектов этот метод возвращает полезный результат, представляющий содержимое объекта. Например, метод `toString()` объекта `java.util.Date` возвращает представляемую им дату, отформатированную в виде строки. Для объектов, не имеющих собственного представления, строковый результат представляет собой обычный идентификатор, который может использоваться для отладки. Если метод `String.valueOf()` вызывается для объекта, то он вызывает метод `toString()` этого объекта и возвращает результат. Между двумя методами есть только одно реальное отличие: при передаче ссылки `null` будет возвращен объект `String` со строкой «null» вместо выдачи исключения `NullPointerException`:

```
Date date = new Date();
// Эквивалент, например "Fri Dec 19 05:45:34 CST 1969"
String d1 = String.valueOf( date );
String d2 = date.toString();

date = null;
d1 = String.valueOf( date ); // "null"
d2 = date.toString(); // NullPointerException!
```

Конкатенация `String` использует метод `valueOf()` во внутренней реализации, так что при «сложении» с помощью оператора `+` с объектом или примитивом вы получите `String`:

```
String today = "Today's date is :" + date;
```

Иногда разработчики используют пустую строку и оператор `+` как сокращенную запись для получения строкового представления объекта, например:

```
String two = "" + 2.384f;
String today = "" + new Date();
```

Сравнение строк

Стандартный метод `equals()` может проверять строки на равенство (содержат ли эти строки одинаковые символы в одинаковом порядке). Для проверки равенства строк без учета регистра используется другой метод, который называется `equalsIgnoreCase()`:

```
String one = "F00";
String two = "foo";
one.equals( two ); // false
one.equalsIgnoreCase( two ); // true
```

Многие начинающие программисты совершают одну и ту же ошибку: для проверки равенства строк они используют оператор `==`, тогда как в действительности им нужен метод `equals()`. Помните, что строки в Java являются объектами, а оператор `==` проверяет *идентичность* объектов (это означает, что два проверяемых аргумента являются одним и тем же объектом). В Java легко создать две строки, которые содержат одинаковые символы, но не являются одним объектом. Пример:

```
String foo1 = "foo";
String foo2 = String.valueOf( new char [] { 'f', 'o', 'o' } );
foo1 == foo2 // false!
foo1.equals( foo2 ) // true
```

Эта ошибка особенно опасна, потому что она часто проявляется при типичных сравнениях строковых литералов (строк, объявленных в двойных кавычках прямо в коде). Дело в том, что Java пытается повысить эффективность работы со строками, объединяя их. Во время компиляции Java находит все идентичные строки внутри класса и создает для них только один объект. Такое объединение безопасно, потому что строки неизменяемы. Вы можете самостоятельно объединять строки подобным образом во время выполнения, используя метод `intern()` класса `String`. *Интернирование* строки возвращает ссылку на эквивалентную строку, уникальную в пределах виртуальной машины.

Метод `compareTo()` сравнивает `String` по лексическому значению с другим объектом `String`, определяя, предшествует ли первая строка второй в порядке алфавитной сортировки, идентична ей или следует после нее. Метод возвращает целое число, которое меньше нуля, равно нулю или больше нуля:

```
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 по позициям их символов в спецификации Unicode. Такой подход работает для простого текста, но не справляется с некоторыми языковыми вариациями. Для более сложных сравнений можно воспользоваться классом `Collator`.

Поиск

Класс `String` предоставляет несколько простых методов для нахождения фиксированных подстрок внутри строки. Методы `startsWith()` и `endsWith()` сравнивают строку-аргумент с началом и концом `String` соответственно:

```
String url = "http://foo.bar.com/";
if ( url.startsWith("http:") ) // true
```

Метод `indexOf()` ищет первое вхождение символа или подстроки; он возвращает начальную позицию найденного вхождения 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
```

Аналогичный метод `lastIndexOf()` проводит поиск в обратном направлении: он ищет последнее вхождение символа или подстроки.

Метод `contains()` решает очень распространенную задачу; он проверяет, содержится ли в строке заданная подстрока:

```
String log = "There is an emergency in sector 7!";
if ( log.contains("emergency") ) pageSomeone();
// Эквивалентно:
if ( log.indexOf("emergency") != -1 ) ...
```

Для расширенного поиска существует API регулярных выражений, позволяющий проводить поиск и разбор по сложным шаблонам. Регулярные выражения будут более подробно рассмотрены далее в этой главе.

Список методов String

В табл. 8.1 перечислены методы, предоставляемые классом `String`. Мы включили в список несколько методов, которые не обсуждались в этой главе, чтобы вы знали другие возможности `String`. Опробуйте эти методы в `jshe11` или просмотрите документацию этого класса на сайте Oracle: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html>.

Таблица 8.1. Методы класса `String`

Метод	Функциональность
<code>charAt()</code>	Получает символ в заданной позиции строки
<code>compareTo()</code>	Сравнивает строку с другой строкой
<code>concat()</code>	Выполняет конкатенацию строки с другой строкой
<code>contains()</code>	Проверяет, содержит ли строка другую строку
<code>copyValueOf()</code>	Возвращает строку, эквивалентную заданному массиву символов
<code>endsWith()</code>	Проверяет, завершается ли строка заданным суффиксом

Таблица 8.1 (окончание)

Метод	Функциональность
<code>equals()</code>	Сравнивает строку с другой строкой
<code>equalsIgnoreCase()</code>	Сравнивает строку с другой строкой без учета регистра
<code>getBytes()</code>	Копирует символы из строки в байтовый массив
<code>getChars()</code>	Копирует символы из строки в символьный массив
<code>hashCode()</code>	Возвращает хеш-код строки
<code>indexOf()</code>	Ищет первое вхождение символа или подстроки в строке
<code>intern()</code>	Получает уникальный экземпляр строки из глобального пула совместно используемых строк
<code>isBlank()</code>	Возвращает <code>true</code> , если строка имеет нулевую длину или содержит только символы-пропуски
<code>isEmpty()</code>	Возвращает <code>true</code> , если строка имеет нулевую длину
<code>lastIndexOf()</code>	Ищет последнее вхождение символа или подстроки в строке
<code>length()</code>	Возвращает длину строки
<code>lines()</code>	Возвращает поток строк, разделенных завершителями строк
<code>matches()</code>	Определяет, совпадает ли вся строка с шаблоном регулярного выражения
<code>regionMatches()</code>	Проверяет, совпадает ли часть строки с заданной частью другой строки
<code>repeat()</code>	Возвращает конкатенацию строки, повторенную заданное количество раз
<code>replace()</code>	Заменяет все вхождения символа в строке другим символом
<code>replaceAll()</code>	Заменяет все совпадения шаблона регулярного выражения другим шаблоном
<code>replaceFirst()</code>	Заменяет первое совпадение шаблона регулярного выражения другим шаблоном
<code>split()</code>	Разбивает строку на массив строк, используя регулярное выражение в качестве разделителя
<code>startsWith()</code>	Проверяет, начинается ли строка с заданного префикса
<code>strip()</code>	Удаляет начальные и конечные пропуски, определяемые вызовом <code>Character.isWhitespace()</code>
<code>stripLeading()</code>	Удаляет начальные пропуски (по аналогии со <code>strip()</code>)
<code>stripTrailing()</code>	Удаляет конечные пропуски (по аналогии со <code>strip()</code>)
<code>substring()</code>	Возвращает подстроку
<code>toArray()</code>	Возвращает массив символов для содержимого строки
<code>toLowerCase()</code>	Преобразует строку к нижнему регистру

Метод	Функциональность
<code>toString()</code>	Возвращает строковое представление объекта
<code>toUpperCase()</code>	Преобразует строку к верхнему регистру
<code>trim()</code>	Удаляет начальные и конечные пропуски, определяемые как символы с кодовым номером, меньшим либо равным 32 (символ «пробел»)
<code>valueOf()</code>	Возвращает строковое представление значения

Создание объектов по строкам

Разбор и форматирование текста — это большая и открытая тема. До сих пор в этой главе рассматривались только элементарные операции со строками: создание, поиск, преобразование простых значений в строки. Теперь пора перейти к более структурированным формам текста. В Java есть богатый набор API для разбора и вывода на экран отформатированных строк, в том числе содержащих текстовые представления чисел, значений даты и времени, денежных сумм. Основную часть этих операций мы рассмотрим прямо сейчас, а форматирование дат и времени отложим до раздела «Локальные дата и время», с. 293.

Начнем с *разбора (parsing)*, или *парсинга*, — чтения примитивных чисел и значений, представленных в виде строк, и разбиения длинных строк на лексемы (tokens). Затем мы рассмотрим регулярные выражения — самое мощное средство разбора текста, существующее в Java. Регулярные выражения позволяют вам определять собственные шаблоны произвольной сложности, использовать их для поиска и выделять их при разборе текста.

Разбор примитивных чисел

В Java числа, символы и логические значения представлены примитивными типами — не объектами. Но для каждого примитивного типа Java также определяет *класс-обертку (primitive wrapper class)*. А именно в пакете `java.lang` есть следующие классы: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` и `Boolean`. Они уже рассматривались в разделе «Обертки для примитивных типов», с. 174, но мы возвращаемся к ним, потому что эти классы содержат статические вспомогательные методы для получения своих типов из строк. В каждом классе-обертке для примитивного типа есть метод парсинга, который читает `String` и возвращает соответствующий примитивный тип. Пример:

```
byte b = Byte.parseByte("16");
int n = Integer.parseInt("42");
long l = Long.parseLong("9999999999");
```

```
float f = Float.parseFloat("4.2");
double d = Double.parseDouble("99.9999999");
boolean b = Boolean.parseBoolean("true");
```

Кроме того, класс `java.util.Scanner` предоставляет единый API, способный не только получать отдельные примитивные значения из строк, но и читать их из потока лексем. Следующий пример показывает, как использовать `Scanner` вместо классов-обертток:

```
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();
```

Разбиение текста на лексемы

Одна из типичных задач программирования — разбор строки текста на слова, или лексемы, между которыми расположены символы-разделители из заданного набора (например, пробелы или запятые). Первый пример содержит слова, разделенные одиночными пробелами. Во втором, более реалистичном примере поля разделены запятыми:

```
Now is the time for all good men (and women)...
```

```
Check Number, Description, Amount
4231, Java Programming, 1000.00
```

В Java есть несколько API (к сожалению, перекрывающихся) для подобных ситуаций. Самые мощные и полезные из них — метод `split()` класса `String` и класс `Scanner`. В обоих API используются регулярные выражения для разбиения строки по произвольным шаблонам. Регулярные выражения еще не рассматривались, но сейчас мы приведем пример с ними, а подробнее объясним далее в этой главе. Кроме того, нам пригодится старый класс `java.util.StringTokenizer`, использующий простые наборы символов для разбиения строк. Он не такой мощный, зато не требует знания синтаксиса регулярных выражений.

Метод `split()` класса `String` получает регулярное выражение, которое описывает разделитель, и использует его для разбиения строки на массив объектов `String`:

```
String text = "Now is the time for all good men";
String [] words = text.split("\\s");
// Слова = "Now", "is", "the", "time", ...

String text = "4231, Java Programming, 1000.00";
String [] fields = text.split("\\s*,\\s*");
// Поля = "4231", "Java Programming", "1000.00"
```


В первом примере используется регулярное выражение `\\s`, которое соответствует одиночному символу пропуска («пустого места»): пробелу, табуляции или возврату курсора. Метод `split()` вернул массив из восьми строк. Во втором примере используется более сложное регулярное выражение `\\s*,\\s*`, которое соответствует запятой, окруженной любым количеством смежных пропусков (хотя их может и не быть). Наш текст сократился до трех удобных аккуратных полей.

С новым API класса `Scanner` можно сделать следующий шаг и разобрать строку с числами из второго примера:

```
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 передаем классу `Scanner` регулярное выражение, которое он должен взять в качестве разделителя, а затем многократно обращаемся к нему с вызовами, чтобы преобразовать все поля в значения соответствующих типов. Класс `Scanner` удобен, потому что он может читать данные не только из `String`, но и непосредственно из источников-поточков (подробнее в главе 11), таких как `InputStream`, `File` и `Channel`:

```
Scanner fileScanner = new Scanner( new File("spreadsheet.csv") );
fileScanner.useDelimiter( "\\s*,\\s*" );
// ...
```

Кроме того, `Scanner` позволяет «заглянуть вперед» при помощи методов вида `hasNext...`, чтобы проверить следующий элемент в источнике данных:

```
while( scanner.hasNextInt() ) {
    int n = scanner.nextInt();
    ...
}
```

StringTokenizer

Уже упомянутый класс `StringTokenizer` сейчас считается устаревшим, но вам лучше о нем знать. Он существует с первых дней Java и часто встречается в коде. `StringTokenizer` позволяет задать разделитель в виде набора символов; любое количество или сочетание этих символов интерпретируется как разделитель между лексемами. Следующий фрагмент читает слова из первого примера:

```
String text = "Now is the time for all good men (and women)...";
StringTokenizer st = new StringTokenizer( text );
while ( st.hasMoreTokens() ) {
    String word = st.nextToken();
    ...
}
```

Методы `hasMoreTokens()` и `nextToken()` вызываются для перебора слов текста. По умолчанию класс `StringTokenizer` использует в качестве разделителей стандартные символы-пропуски: перенос строки, табуляция и т. д. Вы можете задать собственный набор символов-разделителей в конструкторе `StringTokenizer`. Любые комбинации смежных разделителей между лексемами в целевой строке пропускаются:

```
String text = "4231, Java Programming, 1000.00";
StringTokenizer st = new StringTokenizer( text, "," );

while ( st.hasMoreTokens() ) {
    String word = st.nextToken();
    // Слова = "4231", " Java Programming", "1000.00"
}
```

Это решение получается не таким чистым, как в примере с регулярным выражением. Здесь разделителем является запятая, поэтому в поле «Description» появился лишний пробел в начале. Если бы мы добавили пробел в строку разделителя, то класс `StringTokenizer` разбил бы описание на два слова, «Java» и «Programming», а это совсем не то, что требовалось. В данном случае следовало бы вызвать метод `trim()` для удаления начальных и конечных пробелов в каждом элементе.

Регулярные выражения

Пришло время ненадолго отвлечься от изучения Java и прогуляться в страну регулярных выражений. *Регулярное выражение* (*regular expression*, *regex*) — это шаблон для текста. Регулярные выражения используются многими программами (среди которых пакет `java.util.regex`, текстовые редакторы и многие языки сценариев) для сложных операций поиска и манипуляций со строками.

Если вы уже знакомы с концепцией регулярных выражений и их использованием в других языках, этот раздел можно пропустить. Но все равно просмотрите раздел «API регулярных выражений `java.util.regex`», с. 281, — в нем рассматриваются классы Java, предназначенные для работы с регулярными выражениями. А если вы не сталкивались с этой темой и не знаете, что такое регулярные выражения, — откройте свой любимый напиток и приготовьтесь. На следующих страницах вам предстоит знакомство с самым мощным инструментом в арсенале работы с текстом. По сути, это мини-язык внутри языка Java.

Общие сведения о регулярных выражениях

Регулярные выражения описывают текстовые шаблоны специальными символами. Под шаблонами имеются в виду почти любые виды элементов текста,

которые можно идентифицировать только по символам (буквам, цифрам, знакам), независимо от их смысла. Это могут быть отдельные символы, слова, группы слов, строки и абзацы, знаки препинания, регистры букв, а еще чаще — строки и числа с особой структурой (телефонные номера, адреса электронной почты). С помощью регулярных выражений легко найти в словаре все слова, в которые входит буква «Q» без соседней буквы «U», или слова, начинающиеся и заканчивающиеся одной и той же буквой. После составления шаблона вы сможете простыми средствами находить совпадающие с ним фрагменты текста или проверять, какие из заданных строк ему соответствуют. Регулярное выражение можно составить даже так, чтобы оно помогло вам извлечь из текста определенные фрагменты и затем вставить их в нужные места другого текста.

Проще или сложнее?

Прежде чем двигаться дальше, скажем несколько слов о синтаксисе регулярных выражений вообще. В начале этого раздела мы между делом упомянули о том, что будем обсуждать новый язык. Действительно, регулярные выражения представляют собой простую форму языка программирования. Если ненадолго задуматься над примерами, приводившимися ранее, то становится ясно, что для описания даже простых шаблонов (например, адресов электронной почты), которые могут принимать разные формы, потребуется простая разновидность языка программирования.

Книги по компьютерной теории относят регулярные выражения к нижнему уровню иерархии компьютерных языков в отношении как того, что они могут описывать, так и того, что с ними можно делать. Впрочем, это вовсе не означает, что они не могут быть сложными. Как и у большинства языков программирования, элементы регулярных выражений просты, но из них могут строиться сколь угодно сложные выражения. И здесь начинаются проблемы.

Так как регулярные выражения работают со строками, было бы удобно иметь очень компактные обозначения, которые легко вставляются между символами. Но компактная запись может быть очень загадочной, а опыт показывает, что написать сложную команду намного проще, чем прочитать ее позднее. В этом кроется проблема регулярных выражений. Представьте, что поздним вечером в приступе вдохновения, навеянного чашкой кофе, вы написали замечательный однострочный шаблон для сокращения вашей программы. Когда вы вернетесь к чтению этой строки на следующий день, она покажется вам египетскими иероглифами. Обычно чем проще, тем лучше, но если вы можете разбить свою задачу и решить ее более наглядно за несколько этапов, возможно, так и стоит поступить.

Экранирование символов

В общем, мы вас предупредили, но теперь осталось уточнить еще один момент. Мало того что регулярные выражения бывают не для слабонервных, они также

могут приводить к неоднозначной интерпретации обычных строк Java. Важной частью синтаксиса регулярных выражений являются *экранированные символы*. Экранированным называется символ с обратной косой чертой (\) перед ним. Например, экранированный символ \d (обратная косая черта и буква «d») — это сокращение, обозначающее любую цифру (от 0 до 9). Из этого следует, что вы не можете просто записать \d как часть строки Java, потому что (если вы не забыли) в Java символ \ служит для обозначения специальных символов и для определения символьных последовательностей Unicode (\uxxxx). К счастью, в Java предусмотрено решение: экранированный символ обратной косой черты \\ обозначает литеральный символ \.

Таким образом, если вы хотите, чтобы ваше регулярное выражение содержало символ \ (например, \d), его надо экранировать вторым таким же символом:

```
"\\d" // Строка Java для регулярного выражения "любая цифра"
```

Более того, поскольку в синтаксисе регулярных выражений обратная косая черта тоже обозначает специальные символы, она сама требует дополнительного экранирования! И если вы хотите составить регулярное выражение, обозначающее одиночный литеральный символ \, то оно должно выглядеть так:

```
"\\\\\\\" // Строка Java для регулярного выражения "обратная косая черта"
```

Многие «волшебные» операторы, описанные в этом разделе, работают с предшествующим символом, поэтому их также надо экранировать, чтобы использовать их литеральный смысл. К этой категории относятся такие символы, как точка (.), звездочка (*), плюс (+), фигурные скобки { } и круглые скобки ().

Если вы составляете часть выражения с большим количеством литеральных символов, в этом вам помогут специальные разделители \Q и \E. Любой текст, заключенный между \Q и \E, автоматически экранируется. (При этом вы по-прежнему должны использовать экранирование в String: две обратные косые черты обозначают одну; именно две, а не четыре.) Также есть статический метод Pattern.quote(), который делает то же самое и возвращает правильно экранированную версию переданной ему строки.

В остальном мы можем дать только один совет, чтобы сохранить рассудок при работе с этими примерами: храните две копии — строку комментария с базовым регулярным выражением и строку Java, в которой все символы \ удваиваются. И не забудьте про jshell! Это очень полезная «песочница» для тестирования и экспериментов с шаблонами.

Символы и символьные классы

А теперь попрактикуемся в синтаксисе регулярных выражений. Простейшая форма регулярного выражения — это шаблон в виде обычного литерального

текста, не имеющего никакого специального значения. Он проверяется на абсолютное (с точностью до символа) совпадение с какими-либо фрагментами анализируемой строки. Такой шаблон может состоять из одного или нескольких символов. Например, в следующей строке шаблон `s` совпадает с символом `s` в словах `rose` и `is`:

"A rose is \$1.99."

Шаблон `rose` совпадает только со словом `rose`. Впрочем, этот пример вполне очевиден. Гораздо интереснее шаблоны в виде специальных символов и символьных классов.

Любой символ: точка (.)

Шаблон «точка» (.) совпадает с любым одиночным символом. Шаблон `.ose` совпадает со словами `rose`, `nose`, `ose` (пробел, за которым следует `ose`), то есть с сочетанием любого символа и последующими буквами `ose`. Две точки совпадают с двумя любыми символами подряд (`prose`, `close`). Оператор «точка» неразборчив: обычно он останавливается только на символе конца строки (причем вы можете приказать ему, чтобы он и этого не делал; об этом — далее).

Мы можем считать, что точка (.) обозначает символьный класс (то есть группу символов), куда входят все символы. Существуют регулярные выражения и для более интересных символьных классов:

Пропуск: \s, не пропуск: \S

Шаблон `\s` совпадает с любым пропуском (whitespace) в тексте, то есть с пробелом или одним из следующих символов: `\t` (табуляция), `\r` (возврат курсора), `\n` (новая строка), `\f` (прогон страницы). Парный шаблон `\S` инвертирует выбор: он совпадает с любым символом, кроме пропуска.

Цифра: \d, не цифра: \D

Специальный символ `\d` обозначает любую из цифр 0–9. Напротив, `\D` обозначает любой символ, который не является цифрой.

Буквенно-цифровой символ: \w, любой другой символ: \W

Специальный символ `\w` обозначает любой символ из следующих: буквы A–Z, a–z, цифры 0–9, символ подчеркивания (`_`). Напротив, `\W` обозначает любой другой символ.

Нестандартные символьные классы

Вы можете определять собственные символьные классы, используя запись [...]. Например, следующий класс совпадает с любым из символов `a`, `b`, `c`, `x`, `y` или `z`:

[abcxyz]

Специальная запись вида `x-y` используется для обозначения диапазона любых алфавитно-цифровых символов. В следующем примере определяется символьный класс, состоящий из всех букв верхнего и нижнего регистра:

```
[A-Za-z]
```

Если первым символом внутри квадратных скобок является `^` («стрелка вверх»), то символьный класс инвертируется. Следующий шаблон совпадает с любым символом, кроме букв `A–F` верхнего регистра:

```
[^A-F] // G, H, I, ..., a, b, c, ... и так далее.
```

Вложенные символьные классы просто добавляются в перечисление:

```
[A-F[G-Z]\s] // A-Z и пропуск
```

Логическая операция `AND (&&)` вычисляет пересечение (символы, входящие в оба множества):

```
[a-p&&[l-z]] // l, m, n, o, p
[A-Z&&[^P]] // От A до Z, кроме P
```

Маркеры позиции

Шаблон `"[Aa] rose"` (с буквой `a` в верхнем или нижнем регистре) совпадает три раза в следующей фразе:

```
"A rose is a rose is a rose"
```

Позиционные символы позволяют задать относительную позицию совпадения. Самые важные из них, `^` и `$`, обозначают соответственно начало и конец строки:

```
^[Aa] rose // Шаблон совпадает с "A rose" в начале строки
[Aa] rose$ // Шаблон совпадает с "a rose" в конце строки
```

Если говорить конкретнее, `^` и `$` обозначают начало и конец «входных данных» (input), которые часто представляют собой одну строку. Если вы работаете с несколькими строками текста и хотите находить совпадения в начале и в конце отдельных логических строк в пределах одной большой физической строки, включите «многострочный» режим при помощи флага, как описано в разделе «Специальные параметры», с. 280.

Маркеры позиции `\b` и `\B` обозначают границу слова или позицию, не являющуюся границей слова, соответственно. Например, следующий шаблон совпадает с `rose` и `rosemary`, но не с `primrose`:

```
\brose
```

Операторы повторения

Простые фиксированные шаблоны — это только начало. Перейдем к операторам, которые обозначают множественные вхождения символа (или в более общей формулировке — множественные вхождения шаблона, как будет показано в подразделе «Класс Pattern», с. 281):

Ноль или более повторений: звездочка ()*

Оператор «звездочка» (*) после символа или символьного класса обозначает любое количество символов указанного типа — иначе говоря, ноль или более таких символов. Например, следующий шаблон совпадает с цифрой, перед которой есть любое количество начальных нулей (возможно, нет ни одного):

`0*\d` // Любая цифра, перед которой может быть любое количество нулей

Одно или более повторений: знак «плюс» (+)

Оператор «плюс» (+) обозначает одно или несколько повторений. Он эквивалентен выражению `XX*` (шаблону, за которым следует шаблон и звездочка). Например, следующий шаблон обозначает число, состоящее из одной или нескольких цифр, а также необязательных начальных нулей:

`0*\d+` // Число (из одной или нескольких цифр)
// с необязательными начальными нулями

Может показаться, что совпадение с нулями в начале выражения избыточно, потому что ноль — тоже цифра, а следовательно, он все равно совпадает с частью `\d+` выражения. Однако скоро мы покажем, как с помощью регулярных выражений разбирать строки на нужные составляющие. Возможно, в этом случае вы предпочтете удалить начальные нули и оставить только цифры.

Ноль или одно повторение: вопросительный знак (?)

Оператор «вопросительный знак» (?) обозначает ноль повторений или одно повторение. Например, следующий шаблон совпадает с датой истечения срока действия кредитной карты, которая может содержать необязательный символ / в середине:

`\d\d/?\d\d` // Четыре цифры с необязательной косой чертой в середине

*Диапазон (от **x** до **y** повторений включительно): {**x**,**y**}*

Оператор диапазона {**x**,**y**} — самый универсальный оператор повторения. Он задает точный диапазон для количества повторений. Диапазон получает два аргумента (нижнюю и верхнюю границу), разделенные запятой. Следующий шаблон совпадает с любым словом, содержащим от 5 до 7 символов включительно:

`\b\w{5,7}\b` // Слово, содержащее от 5 до 7 символов

Не менее x повторений: {x, }

Если верхняя граница диапазона не указана, а в диапазоне просто стоит запятая, то верхняя граница уходит в бесконечность. Так задается минимальное количество вхождений символа без верхней границы.

Альтернатива

Оператор «вертикальная черта» (|) обозначает логическую операцию OR; он также называется альтернативой. Оператор | не работает с отдельными символами, а применяется ко всему, что находится с любой стороны от него. Он разбивает выражение надвое, если только не будет ограничен группирующими круглыми скобками. Например, слегка наивная попытка разбора даты может выглядеть так:

```
\w+, \w+ \d+, \d+|\d\d/\d\d/\d\d\d\d // Шаблон 1 или шаблон 2
```

В этом выражении левая часть совпадает с такими строками, как *Fri, Oct 12, 2001*, а правая — со строками вида *10/12/2001*.

Следующее регулярное выражение может использоваться для поиска совпадений адресов электронной почты, принадлежащих к одному из трех доменов (*net*, *edu* и *gov*):

```
\w+@[w.]*\.(net|edu|gov) // Адрес электронной почты, завершающийся
// доменом .net, .edu или .gov
```

Специальные параметры

Некоторые специальные параметры управляют тем, как ядро регулярных выражений производит поиск совпадений. Эти параметры могут применяться двумя способами:

- Передача одного или нескольких флагов при вызове метода `Pattern.compile()` (рассматривается в следующем разделе).
- Включение специального блока в регулярное выражение.

В этом разделе будет продемонстрирован второй вариант. Для этого один или несколько флагов заключаются в специальный блок (*?x*), где *x* — флаг включаемого параметра. Обычно этот блок размещается в начале регулярного выражения. Для отключения флагов используется знак «минус» (*?-x*), что позволяет избирательно применять флаги к отдельным частям шаблона.

Доступны следующие флаги:

Поиск совпадения без учета регистра: (?i)

Флаг (*?i*) приказывает ядру регулярных выражений игнорировать регистр символов в процессе поиска. Пример:

```
(?i)yahoo // Совпадает с Yahoo, yahoo, yah00 и т.д.
```


Совпадение точки с любым символом: (?s)

Флаг (?s) включает режим, в котором шаблон «точка» (.) совпадает с любым символом, включая символ конца строки. Данный режим особенно полезен при поиске совпадений, охватывающих несколько логических строк. Имя флага «s» происходит от «single-line mode» (однострочный режим), что выглядит немного странно: этот термин унаследован из языка Perl.

Многострочные якорные символы: (?m)

По умолчанию специальные символы ^ и \$ на самом деле обозначают не начало и конец логических строк (определяемых комбинациями символов возврата курсора или новой строки); вместо этого они обозначают начало или конец всего входного текста. Во многих случаях весь входной текст образует одну строку. Большие блоки текста часто разбиваются на отдельные строки по другим причинам, после чего проверка совпадения в каждой отдельной строке выполняется элементарно, а ^ и \$ ведут себя так, как ожидалось. Но если вы хотите использовать регулярное выражение со всей входной строкой, содержащей несколько логических строк (разделенных комбинациями символов возврата курсора или новой строки), включите многострочный режим при помощи флага (?m). В этом случае специальные символы ^ и \$ будут обозначать начало и конец каждой логической строки блока текста, а также начало и конец всего блока. Точнее говоря, это будет позиция перед первым символом, позиция после последнего символа, а также все позиции до и после завершителей логических строк внутри физической строки.

Строки Unix: (?d)

Флаг (?d) ограничивает определение завершителей строк для специальных символов ^, \$ и . символами новой строки в стиле Unix (\n). По умолчанию также допустимы комбинации «возврат курсора + новая строка» (\r\n).

API регулярных выражений java.util.regex

Итак, мы разобрались с теорией составления регулярных выражений; самое трудное позади. Остается исследовать Java API под названием `java.util.regex`, который нужен для применения этих выражений на практике.

Класс Pattern

Как было сказано ранее, шаблоны регулярных выражений, которые мы записываем в виде строк, на самом деле представляют собой мини-программы, описывающие поиск совпадений в тексте. С помощью пакета регулярных выражений Java компилирует эти маленькие программы в такую форму, которая может применяться к некоторому целевому тексту. Некоторые простые вспомогательные методы получают напрямую строки, используемые в качестве шаблонов.

Однако на более общем уровне Java позволяет явно скомпилировать шаблон и инкапсулировать его в экземпляре класса `Pattern`. Это самый эффективный способ работы с шаблонами, которые могут использоваться многократно, потому что он избавляет от лишних перекомпиляций строк. Шаблоны компилируются статическим методом `Pattern.compile()`:

```
Pattern urlPattern = Pattern.compile("\\w+://[\\w/]*");
```

После того как объект `Pattern` будет создан, вы приказываете ему создать объект `Matcher`, связывающий шаблон с целевой строкой:

```
Matcher matcher = urlPattern.matcher( myText );
```

Объект `Matcher` выполняет поиск совпадений. Вскоре мы поговорим об этом, но сначала стоит упомянуть один вспомогательный метод класса `Pattern`. Этот статический метод `Pattern.matches()` просто получает две строки — регулярное выражение и целевую строку — и определяет, совпадает ли целевая строка с регулярным выражением. Это очень удобно, если вы хотите быстро выполнить однократную проверку в своем приложении. Пример:

```
Boolean match = Pattern.matches( "\\d+\\.\\d+f?", myText );
```

Этот код проверяет, содержит ли строка `myText` число с плавающей точкой в стиле Java (например, `42.0f`). Обратите внимание: совпадением в данном случае считается только полное совпадение строки. Если вы хотите узнать, совпадает ли маленький шаблон с каким-либо фрагментом большой строки (а остальная часть строки вас не интересует), используйте `Matcher`, как описано в разделе «`Matcher`», с. 284.

Теперь составим еще один простой шаблон, который можно будет использовать в нашей игре, когда мы дадим возможность нескольким игрокам соревноваться друг с другом. Многие системы входа используют адреса электронной почты как идентификаторы пользователей. Конечно, такие системы не идеальны, но для наших целей адреса электронной почты отлично подойдут. Программа должна предложить пользователю ввести свой адрес, но нам хотелось бы проверить его на действительность перед использованием. Регулярное выражение позволяет быстро выполнить такую проверку¹.

¹ Задача полной проверки адресов email очень сложна и выходит далеко за рамки темы регулярных выражений. Регулярные выражения могут проверять большинство адресов email, но если такая проверка нужна вам для коммерческого или другого профессионального приложения, лучше воспользуйтесь сторонними библиотеками, например классом `EmailValidator` из проекта Apache Commons (<https://commons.apache.org/proper/commons-validator/apidocs/org/apache/commons/validator/routines/EmailValidator.html>).

По аналогии с тем, как вы пишете алгоритмы для решения задач программирования, проектирование регулярного выражения требует разбиения задачи поиска совпадения шаблона на меньшие, более простые части. Если мы рассматриваем адреса электронной почты, нетрудно заметить в их структуре некоторые закономерности. Самое очевидное — это символ @ в середине некоторых адресов. Наивный шаблон (но лучше такой, чем никакого!), основанный на этом факте, составляется следующим образом:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches(".*@.*", sample);
```

Но этот шаблон слишком неразборчив. Безусловно, он определит многие действительные адреса электронной почты, но также пропустит и множество недопустимых вида "bad.address@", "@also.bad" или даже "@@". (Проверьте их в `js` и даже придумайте пару собственных неудачных примеров.) Как повысить качество поиска? Одно простое улучшение — использование модификатора `+` вместо `*`. Обновленный шаблон теперь требует, чтобы с каждой стороны от @ присутствовал хотя бы один символ. Однако это не все, что мы знаем об адресах электронной почты. Например, левая «половина» адреса (имя) не может содержать символ @ — как не может его быть и в доменной части. Для дальнейшего усовершенствования можно воспользоваться нестандартным символьным классом:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+", sample);
```

Этот шаблон уже лучше, но он все еще допускает такие ошибочные адреса, как `still@bad`, потому что доменные имена состоят как минимум из имени, за которым следует точка (.) и домен верхнего уровня (TLD), например `oreilly.com`. Таким образом, шаблон приходит к следующему виду:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+\.\.(com|org)", sample);
```

Этот шаблон решает проблему с адресами вида `still@bad`, но мы, пожалуй, зашли слишком далеко не в ту сторону. Существует очень много разных TLD — слишком много, чтобы составить их список, даже если забыть о проблеме сопровождения этого списка при добавлении новых TLD¹. Немного отступим и оставим точку в доменной части, но удалим конкретные TLD, чтобы допустимой считалась любая последовательность букв:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+\.[a-z]+", sample);
```

¹ Вы даже можете зарегистрировать собственный глобальный TLD, если у вас найдется несколько лишних сотен тысяч долларов.

Намного лучше. Добавим последнее улучшение, чтобы не приходилось беспокоиться о регистре символов, так как в адресах электронной почты регистр игнорируется. Для этого просто добавим флаг:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("(?i)[^@]+@[^@]+\\.([a-z])+", sample);
```

Еще раз подчеркнем, что это регулярное выражение ни в коем случае не идеально. Но оно станет хорошей отправной точкой, и его будет достаточно для нашей простой системы входа в приложение после добавления сетевой поддержки. Если вы захотите немного поэкспериментировать с шаблоном, расширить или улучшить его, не забывайте, что в `jshell` можно повторно использовать строки при помощи клавиш управления курсором. Клавиша `↑` загружает предыдущую строку. Клавишами `↑` и `↓` можно переходить между всеми недавно выполнявшимися строками. Внутри строки используйте клавиши `←` и `→` для перемещения, а также для удаления, добавления и редактирования команд. Затем нажмите клавишу `Enter` (`Return`), чтобы выполнить измененную команду, — вам не обязательно перемещать курсор в конец строки перед нажатием `Enter` (`Return`).

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.([a-z])+", "good@some.domain")
$1 ==> true
```

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.([a-z])+", "good@oreilly.com")
$2 ==> true
```

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.([a-z])+", "oreilly.com")
$3 ==> false
```

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.([a-z])+", "bad@oreilly@com")
$4 ==> false
```

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.([a-z])+", "me@oreilly.COM")
$5 ==> true
```

```
jshell> Pattern.matches("[^@]+@[^@]+\\.([a-z])+", "me@oreilly.COM")
$6 ==> false
```

В этих примерах мы ввели полную строку `Pattern.matches(...)` всего один раз. После этого нам оставалось только нажимать `↑`, редактировать строку, а затем нажимать `Enter` (`Return`) в следующих пяти строках. Кстати, вы видите, почему последняя проверка завершилась неудачей?

Matcher

Объект `Matcher` связывает шаблон со строкой и предоставляет средства для проверки, поиска и перебора совпадений шаблона. `Matcher` обладает *состоянием*.

Например, метод `find()` пытается найти следующее совпадение при каждом вызове. При этом вы можете сбросить `Matcher` и начать все заново, вызвав метод `reset()`.

Если вас интересует только «одно большое совпадение», то есть ожидается, что строка либо совпадает с шаблоном, либо нет, используйте методы `matches()` или `lookingAt()`. Они приблизительно соответствуют методам `equals()` и `startsWith()` класса `String`. Метод `matches()` проверяет, совпадает ли строка с шаблоном полностью (без каких-либо лишних символов), и возвращает `true` или `false`. Метод `lookingAt()` делает то же самое, за исключением того, что он только проверяет, начинается ли строка с совпадения шаблона, и не проверяет, используются ли все символы строки.

Часто требуется провести поиск по строке и найти одно или несколько совпадений. Для этого можно воспользоваться методом `find()`. Каждый вызов `find()` возвращает `true` или `false` для следующего совпадения шаблона и сохраняет во внутреннем представлении позиции совпавшего текста. Вы можете получить позиции начального и конечного символов методами объекта `Matcher` с именами `start()` и `end()`, а также просто получить совпадающий текст методом `group()`. Пример:

```
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() );
```

Этот фрагмент выводит начальные позиции 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 совпадающего текста называется `group()`, потому что он относится к нулевой *захватывающей группе* (*capture group*), означающей полное совпадение. Также можно получить текст других нумерованных захватывающих групп, вызывая метод `group()` с целочисленным аргументом. Для получения общего количества сохраняющих групп есть метод `groupCount()`:

```
for (int i=1; i < matcher.groupCount(); i++)
    System.out.println( matcher.group(i) );
```

Разбиение строк на лексемы

Очень распространенная задача — разбиение строки на поля по некоторому разделителю (например, по запятой). Задача встречается настолько часто, что в классе `String` для нее есть специальный метод. Метод `split()` получает регулярное выражение и возвращает массив подстрок, полученный разбиением по этому шаблону. Возьмем следующую строку и вызовы `split()`:

```
String text = "Foo, bar ,   blah";
String[] badFields = text.split(",");
String[] goodFields = text.split( "\\s*,\\s*" );
```

Первый вызов `split()` возвращает массив `String`, но наивное использование символа «`,`» для разбиения строки означает, что все пробелы в тексте будут «прилипать» к более интересным символам. Как и ожидалось, `Foo` извлекается как отдельное слово, но затем идет `bar` *<пробел>* и, наконец, *<пробел>* *<пробел>* *<пробел>* `blah`. Нехорошо! Второй вызов `split()` также дает массив `String`, но на этот раз с ожидаемыми элементами `Foo`, `bar` (без пробелов в конце) и `blah` (без пробелов в начале).

Если вы собираетесь выполнить такую операцию более одного-двух раз, лучше скомпилировать шаблон и использовать его метод `split()`, идентичный его версии из `String`. Метод `split()` класса `String` эквивалентен следующему:

```
Pattern.compile(pattern).split(string);
```

Программисту желательно знать регулярные выражения как можно лучше, не ограничиваясь теми конкретными возможностями их применения в Java, о которых мы рассказали. Вернитесь к `jshell` («Класс `Pattern`», с. 281), чтобы поэкспериментировать с выражениями и разбиением строк. Безусловно, эту тему лучше изучать на практике.

Математические средства

В Java есть встроенная поддержка арифметических операций — как целочисленных, так и с плавающей точкой. А для математических операций более высокого уровня существует класс `java.lang.Math`. Вероятно, вам уже знакомы классы-обертки для примитивных типов данных, которые позволяют интерпретировать их как объекты. Классы-обертки также содержат методы для базовых преобразований типов.

Сначала скажем несколько слов о встроенных арифметических операциях. Java обрабатывает ошибки в целочисленной арифметике выдачей исключения `ArithmeticException`:

```
int zero = 0;

try {
    int i = 72 / zero;
} catch ( ArithmeticException e ) {
    // Деление на ноль
}
```

Чтобы сгенерировать ошибку в этом примере, мы создали промежуточную переменную `zero`. Компилятор довольно изобретателен, и он обнаружил бы попытку прямолинейного деления на ноль-литерал.

С другой стороны, арифметические выражения с плавающей точкой исключений не выдают. Вместо этого они возвращают специальные значения, перечисленные в табл. 8.2.

Таблица 8.2. Специальные значения с плавающей точкой

Значение	Математическое представление
POSITIVE_INFINITY (положительная бесконечность)	1.0 / 0.0
NEGATIVE_INFINITY (отрицательная бесконечность)	-1.0 / 0.0
NaN (не число)	0.0 / 0.0

В следующем примере генерируется бесконечно большой результат:

```
double zero = 0.0;
double d = 1.0/zero;
if ( d == Double.POSITIVE_INFINITY )
    System.out.println( "Division by zero" );
```

Специальное значение NaN («не число») обозначает результат деления ноля на ноль. NaN обладает особым математическим свойством: оно не равно самому себе (`NaN != NaN` дает значение `true`). Для проверки значения на равенство NaN используется метод `Float.isNaN()` или `Double.isNaN()`.

Класс `java.lang.Math`

Класс `java.lang.Math` содержит математическую библиотеку Java. Он содержит набор статических методов для всех основных математических операций, таких как `sin()`, `cos()` и `sqrt()`. Класс `Math` нельзя назвать вполне объектно-ориентированным (вы не сможете создать экземпляр `Math`). Зато он является удобным хранилищем для статических методов, которые больше напоминают глобальные функции. Как мы показали в главе 5, функциональность статического импор-

тирования можно использовать для введения имен таких статических методов и констант прямо в область видимости нашего класса для работы с ними под простыми именами без уточнений (unqualified names).

В табл. 8.3 перечислены методы класса `java.lang.Math`.

Таблица 8.3. Методы `java.lang.Math`

Метод	Тип(ы) аргументов	Функциональность
<code>Math.abs(a)</code>	<code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>	Абсолютное значение
<code>Math.acos(a)</code>	<code>double</code>	Арккосинус
<code>Math.asin(a)</code>	<code>double</code>	Арсинус
<code>Math.atan(a)</code>	<code>double</code>	Арктангенс
<code>Math.atan2(a, b)</code>	<code>double</code>	Угловая часть преобразования декартовых координат в полярные
<code>Math.ceil(a)</code>	<code>double</code>	Наименьшее целое число, большее либо равное a
<code>Math.cbrt(a)</code>	<code>double</code>	Кубический корень
<code>Math.cos(a)</code>	<code>double</code>	Косинус
<code>Math.cosh(a)</code>	<code>double</code>	Гиперболический косинус
<code>Math.exp(a)</code>	<code>double</code>	Число e (основание натурального логарифма) в степени a
<code>Math.floor(a)</code>	<code>double</code>	Наибольшее целое число, меньшее либо равное a
<code>Math.hypot(a, b)</code>	<code>double</code>	Точное вычисление квадратного корня из (a ² + b ²)
<code>Math.log(a)</code>	<code>double</code>	Натуральный логарифм
<code>Math.log10(a)</code>	<code>double</code>	Десятичный логарифм
<code>Math.max(a, b)</code>	<code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>	Наибольшее из двух чисел
<code>Math.min(a, b)</code>	<code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>	Наименьшее из двух чисел
<code>Math.pow(a, b)</code>	<code>double</code>	a в степени b
<code>Math.random()</code>	нет аргумента	Случайное число от 0 до 1
<code>Math rint(a)</code>	<code>double</code>	Преобразование значения <code>double</code> в целочисленное значение в формате <code>double</code>
<code>Math.round(a)</code>	<code>float</code> , <code>double</code>	Округление до целого числа
<code>Math.signum(a)</code>	<code>double</code> , <code>float</code>	Знак числа: 1.0, -1.0 или 0

Метод	Тип(ы) аргументов	Функциональность
<code>Math.sin(a)</code>	<code>double</code>	Синус
<code>Math.sinh(a)</code>	<code>double</code>	Гиперболический синус
<code>Math.sqrt(a)</code>	<code>double</code>	Квадратный корень
<code>Math.tan(a)</code>	<code>double</code>	Тангенс
<code>Math.tanh(a)</code>	<code>double</code>	Гиперболический тангенс
<code>Math.toDegrees(a)</code>	<code>double</code>	Преобразование радианов в градусы
<code>Math.toRadians(a)</code>	<code>double</code>	Преобразование градусов в радианы

Методы `log()`, `pow()` и `sqrt()` могут выдавать исключение времени выполнения `ArithmeticException`. Методы `abs()`, `max()` и `min()` перегружены для всех скалярных значений `int`, `long`, `float` и `double`, то есть возвращают соответствующие типы. Версии `Math.round()` получают либо `float`, либо `double` и возвращают `int` или `long` соответственно. Остальные методы работают со значениями `double` и возвращают значения `double`:

```
double irrational = Math.sqrt( 2.0 ); // 1.414...
int bigger = Math.max( 3, 4 ); // 4
long one = Math.round( 1.125798 ); // 1
```

Просто для того, чтобы подчеркнуть важность статического импортирования, опробуем эти простые функции в `jshell`:

```
jshell> import static java.lang.Math.*

jshell> double irrational = sqrt(2.0)
irrational ==> 1.4142135623730951

jshell> int bigger = max(3,4)
bigger ==> 4

jshell> long one = round(1.125798)
one ==> 1
```

`Math` также содержит статические значения `E` и `PI` в формате `double`:

```
double circumference = diameter * Math.PI;
```

Math в действии

Класс `Math` и его статические методы кратко упоминались в разделе «Обращение к полям и методам», с. 158. Мы снова им воспользуемся, чтобы сделать игру немного интереснее: будем выбирать места появления деревьев случайным

образом. Метод `Math.random()` возвращает случайное значение `double`, большее либо равное 0 и меньшее 1. Добавьте немного математики, округлений — и это значение можно будет использовать для создания случайных чисел в любом нужном диапазоне. Преобразование значения к заданному диапазону выполняется по следующей формуле:

```
int randomValue = min + (int)(Math.random() * (max - min));
```

Попробуйте сами! Попробуйте сгенерировать случайное число из четырех цифр в `jshell`. Присвойте переменной `min` значение 1000, а переменной `max` — 10000:

```
jshell> int min = 1000
min ==> 1000

jshell> int max = 10000
max ==> 10000

jshell> int fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 9603

jshell> fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 9178

jshell> fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 3789
```

Для размещения деревьев нам понадобятся два случайных числа: для координат `x` и `y`. Установим диапазон, при котором деревья будут полностью помещаться на экране; для этого нужно предусмотреть отступы у краев. Для координаты `x` одно из возможных решений может выглядеть так:

```
private int goodX() {
    // Не менее половины ширины дерева плюс еще несколько пикселей
    int leftMargin = Field.TREE_WIDTH_IN_PIXELS / 2 + 5;
    // Вычисляем правый отступ
    int rightMargin = FIELD_WIDTH - leftMargin;

    // Получаем случайное число между левым и правым отступом
    return leftMargin + (int)(Math.random() * (rightMargin - leftMargin));
}
```

Создайте аналогичный метод для нахождения `y`, и вы увидите на экране что-то похожее на рис. 8.1. Можно даже действовать более изощренно и воспользоваться методом `isTouching()`, рассмотренным в главе 5, чтобы избежать размещения деревьев, соприкасающихся с физиком. Обновленный цикл инициализации деревьев выглядит так:

```
for (int i = field.trees.size(); i < Field.MAX_TREES; i++) {
    Tree t = new Tree();
    t.setPosition(goodX(), goodY());
}
```

```
// Деревья могут находиться вблизи друг от друга и перекрываться,  
// но они не должны пересекаться с физиком  
while(player1.isTouching(t)) {  
    // Обнаружено пересечение, пробуем снова  
    t.setPosition(goodX(), goodY());  
    System.err.println("Repositioning an intersecting tree...");  
}  
field.addTree(t);  
}
```

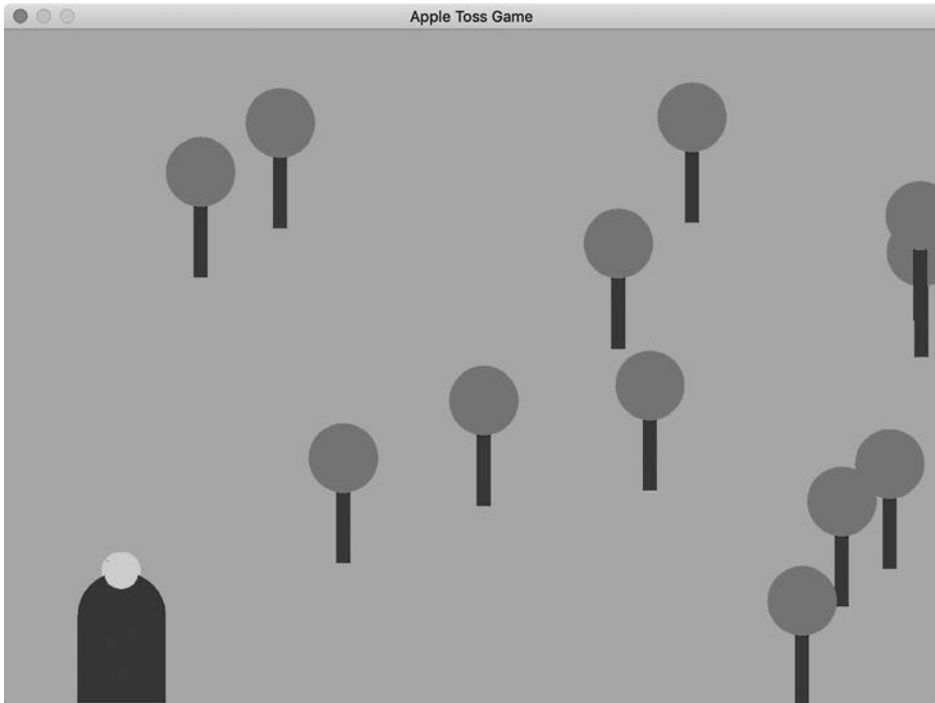


Рис. 8.1. Случайное размещение деревьев

Попробуйте выйти из игры и запустить ее снова. Деревья будут располагаться по-разному при каждом запуске приложения.

Большие числа

Если типы `long` и `double` недостаточно велики или не обеспечивают достаточной точности для ваших целей, пакет `java.math` предоставляет классы `BigInteger` и `BigDecimal`, поддерживающие числа с произвольной точностью. Эти полно-

функциональные классы содержат методы для выполнения математических операций произвольной точности и для точного управления округлением остатков. В следующем примере класс `BigDecimal` используется для сложения двух очень больших чисел и затем для деления с точностью до 100 знаков:

```
long l1 = 9223372036854775807L; // Long.MAX_VALUE
long l2 = 9223372036854775807L;
System.out.println( l1 + l2 ); // -2 ! Нехорошо!

try {
    BigDecimal bd1 = new BigDecimal( "9223372036854775807" );
    BigDecimal bd2 = new BigDecimal( 9223372036854775807L );
    System.out.println( bd1.add( bd2 ) ); // 18446744073709551614

    BigDecimal numerator = new BigDecimal(1);
    BigDecimal denominator = new BigDecimal(3);
    BigDecimal fraction =
        numerator.divide( denominator, 100, BigDecimal.ROUND_UP );
    // Результат деления с точностью до 100 знаков = 0.333333 ... 3334
}
catch (NumberFormatException nfe) { }
catch (ArithmeticException ae) { }
```

Если вы просто ради интереса занимаетесь криптографическими или научными алгоритмами, то класс `BigInteger` вам абсолютно необходим. В свою очередь, класс `BigDecimal` обычно встречается в приложениях, работающих с денежными суммами и финансовыми данными. В других областях эти классы вам вряд ли понадобятся.

Дата и время

Работать со значениями даты и времени, не имея подходящих средств, было бы непросто. До выхода Java 8 в распоряжении программиста имелись только три класса, выполняющих за него большую часть этой работы. Класс `java.util.Date` инкапсулирует момент времени. Класс `java.util.GregorianCalendar`, расширяющий абстрактный класс `java.util.Calendar`, выполняет преобразование между моментом времени и календарными полями (такими, как месяц, день и год). Класс `java.text.DateFormat` умеет генерировать и разбирать строковые представления даты и времени во многих языках.

Хотя классов `Date` и `Calendar` было достаточно для многих практических ситуаций, им не хватало детальности и некоторых функций. Это привело к созданию сторонних библиотек, упростивших работу с датами, моментами времени и интервалами времени. В Java 8 появился пакет `java.time`, где реализованы многие необходимые улучшения в этой области. Мы изучим этот новый пакет, но на практике код с классами `Date` и `Calendar` все еще часто встречается, поэтому

вам будет полезно помнить об их существовании. Как обычно, электронная документация (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Date.html>) является бесценным источником для анализа тех частей Java, которые не рассматриваются в книге.

Локальные дата и время

Класс `java.time.LocalDate` представляет дату без информации о времени для вашего региона (например, 4 мая 2019 года). Подобным образом класс `java.time.LocalTime` представляет время без информации о дате (например, ваш будильник срабатывает в 7:15 каждое утро). Класс `java.time.LocalDateTime` хранит и дату, и время (например, назначенный момент визита к окулисту, который поможет вам углубиться в чтение книг о Java). Все эти классы предоставляют статические методы для создания новых экземпляров: либо вызовом `of()` с передачей подходящих числовых значений, либо вызовом `parse()` с разбором строк. Откройте `jshell` и попробуйте создать несколько примеров.

```
jshell> import java.time.*

jshell> LocalDate.of(2019,5,4)
$2 ==> 2019-05-04

jshell> LocalDate.parse("2019-05-04")
$3 ==> 2019-05-04

jshell> LocalTime.of(7,15)
$4 ==> 07:15

jshell> LocalTime.parse("07:15")
$5 ==> 07:15

jshell> LocalDateTime.of(2019,5,4,7,0)
$6 ==> 2019-05-04T07:00

jshell> LocalDateTime.parse("2019-05-04T07:15")
$7 ==> 2019-05-04T07:15
```

Еще один замечательный статический метод для создания таких объектов, `now()`, предоставляет текущую дату и (или) время:

```
jshell> LocalTime.now()
$8 ==> 15:57:24.052935

jshell> LocalDate.now()
$9 ==> 2019-12-12

jshell> LocalDateTime.now()
$10 ==> 2019-12-12T15:57:37.909038
```

Отлично! После импортирования пакета `java.time` вы можете создавать экземпляры всех классов `Local...` для конкретного момента времени или для момента «прямо сейчас». Возможно, вы заметили, что объекты, создаваемые методом `now()`, включают секунды и наносекунды. Вы можете передать эти значения методам `of()` и `parse()`, если они вам нужны. Ничего особенно эффектного, но когда у вас появятся эти объекты, вы сможете сделать с ними много полезного.

Создание и обработка значений даты и времени

Одним из главных достоинств классов `java.time` является целостный набор методов для сравнения и изменения значений даты и времени. Например, во многих чатах выводится информация о том, сколько времени прошло с момента отправки сообщения. Подпакет `java.time.temporal` содержит именно то, что нам для этого могло бы понадобиться: интерфейс `ChronoUnit`. Он содержит различные единицы даты и времени: `MONTHS`, `DAYS`, `HOURS`, `MINUTES` и т. д. Эти единицы могут использоваться при вычислении разности. Например, при помощи метода `between()` можно вычислить, сколько времени у нас займет создание двух значений даты-времени в `jshell`:

```
jshell> LocalDateTime first = LocalDateTime.now()
first ==> 2019-12-12T16:03:21.875196

jshell> LocalDateTime second = LocalDateTime.now()
second ==> 2019-12-12T16:03:33.175675

jshell> import java.time.temporal.*

jshell> ChronoUnit.SECONDS.between(first, second)
$12 ==> 11
```

Наглядная проверка показывает, что для ввода строки, в которой была создана вторая переменная, потребовалось около 11 секунд. За полным списком доступных единиц обращайтесь к документации `ChronoUnit`, а мы лишь скажем, что в вашем распоряжении полный диапазон: от наносекунд до тысячелетий.

Эти единицы также помогают оперировать с датой и временем методами `plus()` и `minus()`. Например, следующий фрагмент устанавливает напоминание на неделю вперед от сегодняшнего дня:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-12

jshell> LocalDate reminder = today.plus(1, ChronoUnit.WEEKS)
reminder ==> 2019-12-19
```

Неплохо! Но этот пример напоминает еще одну операцию, которую требуется периодически выполнять. Возможно, вы захотите назначить напоминание на

конкретное время 19-го числа. Преобразования между датой, временем и датой-временем достаточно легко выполняется методами `atDate()` и `atTime()`:

```
jshell> LocalDateTime betterReminder = reminder.atTime(LocalTime.of(9,0))
betterReminder ==> 2019-12-19T09:00
```

Теперь вы получите напоминание в 9 утра... Но что, если вы установили напоминание в Атланте, а затем перелетели в Сан-Франциско? Когда сработает сигнал? Значение `LocalDateTime` локально, как следует из имени! Таким образом, часть `T09:00` все равно будет представлять 9 утра, где бы вы ни запускали программу. Но если вы ведете общий календарь или планируете встречу в Сети, игнорировать различия в часовых поясах уже нельзя. К счастью, пакет `java.time` позаботился и об этом.

Часовые пояса

Авторы нового пакета `java.time` рекомендуют при каждой возможности использовать локальные разновидности классов времени и даты. Добавление поддержки часовых поясов означает повышение сложности вашего приложения — они хотят по возможности оградить вас от этой сложности. Однако во многих ситуациях поддержка часовых поясов неизбежна. Для работы с «поясными» датами и временем используются классы `ZonedDateTime` и `OffsetDateTime`. Они поддерживают имена часовых поясов и такие концепции, как летнее время. Класс `OffsetDateTime` представляет постоянное, простое числовое смещение от времени UTC (от гринвичского времени).

В большинстве форматов даты-времени, ориентированных на пользователя, используются именованные часовые пояса, поэтому для начала посмотрим, как создается значение даты-времени с часовым поясом. Для присоединения часового пояса используется класс `ZoneId`, содержащий популярный статический метод `of()` для создания новых экземпляров. Нужное значение в виде `String` передается в виде `String`:

```
jshell> LocalDateTime piLocal = LocalDateTime.parse("2019-03-14T01:59")
piLocal ==> 2019-03-14T01:59
jshell> ZonedDateTime piCentral = piLocal.atZone(ZoneId.of("America/Chicago"))
piCentral ==> 2019-03-14T01:59-05:00[America/Chicago]
```

После этого, например, вы сможете пригласить ваших парижских друзей на встречу в нужное время, и в этом вам поможет метод с длинным, но подходящим именем `withZoneSameInstant()`:

```
jshell> ZonedDateTime piAlaMode =
piCentral.withZoneSameInstant(ZoneId.of("Europe/Paris"))
piAlaMode ==> 2019-03-14T07:59+01:00[Europe/Paris]
```

Если вы хотите пригласить и других друзей, которые живут в регионе, где нет крупного города, используйте метод `systemDefault()` класса `ZoneID` для программного выбора часового пояса:

```
jshell> ZonedDateTime piOther =  
piCentral.withZoneSameInstant(ZoneId.systemDefault())  
piOther ==> 2019-03-14T02:59:04:00[America/New_York]
```

В этом случае `jshell` выполняется на ноутбуке в стандартном восточном часовом поясе США (без учета летнего времени), и значение `piOther` точно совпадает с ожидаемым. Идентификатор часового пояса `systemDefault()` чрезвычайно удобен для быстрого преобразования даты-времени из другого пояса в соответствии с информацией, которую вы с большой вероятностью получите от часов и календаря вашего компьютера. В коммерческих приложениях можно разрешить пользователю выбрать нужный пояс, но `systemDefault()` обычно работает достаточно хорошо.

Разбор и форматирование даты и времени

Для создания и вывода даты-времени (как локальных, так и с часовым поясом) мы пока пользовались стандартными форматами, которые подчиняются стандартам ISO и обычно работают повсюду, где требуется ввести или вывести дату и время. Но как знает каждый программист, «обычно» — не то же самое, что «всегда». К счастью, вы можете воспользоваться вспомогательным классом `java.time.format.DateTimeFormatter`; он поможет вам как при разборе ввода, так и при форматировании вывода.

Центральное место в функциональности `DateTImeFormatter` занимает построение форматной строки, управляющей как разбором, так и форматированием. Форматная строка строится из частей, перечисленных в табл. 8.4. Здесь приводится только часть поддерживаемых параметров, но при их помощи вы справитесь с большинством значений даты и времени, которые вам могут встретиться. Обратите внимание: у этих символов учитывается регистр!

Таблица 8.4. Популярные элементы `DateTImeFormatter`

Символ	Описание	Пример
y	Год	2004; 04
M	Месяц	7; 07
L	Месяц	Jul; July; J
d	День месяца	10

Символ	Описание	Пример
E	День недели	Tue; Tuesday; T
a	Половина суток AM/PM	PM
h	Час в 12-часовом формате (1–12)	12
K	Час в 12-часовом формате (0–11)	0
k	Час в 24-часовом формате (1–24)	24
H	Час в 24-часовом формате (0–23)	0
m	Минуты	30
s	Секунды	55
S	Доли секунды	033954
z	Имя часового пояса	Pacific Standard Time; PST
Z	Смещение часового пояса	+0000; -0800; -08:00

Например, чтобы собрать из этих элементов стандартный короткий формат даты США, используйте символы `M`, `d` и `y`. Форматировщик создается статическим методом `ofPattern()`, после чего он может использоваться (и повторно использоваться) с методом `parse()` любого из классов даты и времени:

```
jshell> import java.time.format.DateTimeFormatter

jshell> DateTimeFormatter shortUS = DateTimeFormatter.ofPattern("MM/dd/yy")
shortUS ==> Value(MonthOfYe ... (YearOfEra,2,2,2000-01-01)

jshell> LocalDate valentines = LocalDate.parse("02/14/19", shortUS)
valentines ==> 2019-02-14

jshell> LocalDate piDay = LocalDate.parse("03/14/19", shortUS)
piDay ==> 2019-03-14
```

Как упоминалось ранее, форматировщик работает в обоих направлениях. Просто используйте метод `format()` вашего форматировщика для получения строкового представления даты или времени:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-14

jshell> shortUS.format(today)
$30 ==> "12/14/19"

jshell> shortUS.format(piDay)
$31 ==> "03/14/19"
```

Конечно, форматировщики работают и со значениями времени, а также даты-времени:

```
jshell> DateTimeFormatter military = DateTimeFormatter.ofPattern("HHmm")
military ==> Value(HourOfDay,2)Value(MinuteOfHour,2)

jshell> LocalTime sunset = LocalTime.parse("2020", military)
sunset ==> 20:20

jshell> DateTimeFormatter basic = DateTimeFormatter.ofPattern("h:mm a")
basic ==> Value(ClockHourOfAmPm)':'Value(MinuteOfHour,2)' 'Text(AmPmOfDay,SHORT)

jshell> basic.format(sunset)
$42 ==> "8:20 PM"

jshell> DateTimeFormatter appointment =
DateTimeFormatter.ofPattern("h:mm a MM/dd/yy z")
appointment ==>
Value(ClockHourOfAmPm)':' ...
0-01-01)' 'ZoneText(SHORT)

jshell> ZonedDateTime dentist =
ZonedDateTime.parse("10:30 AM 11/01/19 EST", appointment)
dentist ==> 2019-11-01T10:30-04:00[America/New_York]

jshell> ZonedDateTime nowEST = ZonedDateTime.now()
nowEST ==> 2019-12-14T09:55:58.493006-05:00[America/New_York]

jshell> appointment.format(nowEST)
$47 ==> "9:55 AM 12/14/19 EST"
```

Обратите внимание, что в двух фрагментах с `ZonedDateTime` мы вывели в конце строк название часового пояса (оно обозначается символом `z`); вероятно, вы не ожидали его там увидеть. Мы хотели показать всю мощь этих форматов. Вы можете создать формат для очень широкого диапазона входных данных или стилей вывода. Старые данные и плохо спроектированные веб-формы — лишь два примера тех областей, в которых `DateTimeFormatter` поможет вам привести все в порядок.

Разбор ошибок

Даже если в вашем распоряжении вся мощь средств разбора, иногда возникают проблемы. К сожалению, исключения, которые при этом выдаются, часто выглядят слишком невразумительно. Возьмем следующую попытку разобрать время с часами, минутами и секундами:

```
jshell> DateTimeFormatter withSeconds = DateTimeFormatter.ofPattern("hh:mm:ss")
withSeconds ==>
Value(ClockHourOfAmPm,2)':' ...
```

```
Value(SecondOfMinute,2)
```

```
jshell> LocalDateTime.parse("03:14:15", withSeconds)
| Exception java.time.format.DateTimeParseException:
| Text '03:14:15' could not be parsed: Unable to obtain
| LocalDateTime from TemporalAccessor: {MinuteOfHour=14, MilliOfSecond=0,
| SecondOfMinute=15, NanoOfSecond=0, HourOfAmPm=3,
| MicroOfSecond=0}, ISO of type java.time.format.Parsed
|   at DateTimeFormatter.createError (DateTimeFormatter.java:2020)
|   at DateTimeFormatter.parse (DateTimeFormatter.java:1955)
|   at LocalDateTime.parse (LocalDateTime.java:463)
|   at (#33:1)
| Caused by: java.time.DateTimeException:
| Unable to obtain LocalDateTime from ...
|   at LocalDateTime.from (LocalDateTime.java:431)
|   at Parsed.query (Parsed.java:235)
|   at DateTimeFormatter.parse (DateTimeFormatter.java:1951)
|   ...
```

Исключение `DateTimeParseException` выдается в любой ситуации, когда не удастся разобрать входную строку. Также оно выдается в таких случаях, как в приведенном примере: поля были разобраны правильно, но не содержали достаточной информации для создания объекта `LocalTime`. Может, это не очевидно, но время «3:14:15» наступает как днем, так и очень ранним утром. Как выясняется, проблема возникла из-за шаблона `hh`. Для ее решения можно либо выбрать шаблон с однозначным 24-часовым диапазоном, либо явно добавить элемент `AM/PM`:

```
jshell> DateTimeFormatter valid1 = DateTimeFormatter.ofPattern("hh:mm:ss a")
valid1 ==> Value(ClockHourOfAmPm,
2)':'Value(MinuteOfHour,2)' ... 2)' 'Text(AmPmOfDay,SHORT)

jshell> DateTimeFormatter valid2 = DateTimeFormatter.ofPattern("HH:mm:ss")
valid2 ==> Value(HourOfDay,2)':'Value(MinuteOfHour,2)':'Value(SecondOfMinute,2)

jshell> LocalDateTime piDay1 = LocalDateTime.parse("03:14:15 PM", valid1)
piDay1 ==> 15:14:15

jshell> LocalDateTime piDay2 = LocalDateTime.parse("03:14:15", valid2)
piDay2 ==> 03:14:15
```

Таким образом, если вы получаете `DateTimeParseException`, но ваши входные данные вроде бы точно соответствуют формату, лишний раз проверьте, что сам формат включает все необходимое для создания даты или времени. И еще одно замечание по поводу этих исключений: для разбора значений годов вам может понадобиться символ «u» — не вполне очевидный по смыслу.

При работе с `DateTimeFormatter` необходимо учитывать еще очень много подробностей! В этом случае электронная документация (<https://docs.oracle.com/en/>)

java/javase/11/docs/api/java.base/java/time/format/DateTimeFormatter.html) пригодится вам больше, чем для многих других вспомогательных классов.

Временные метки

Еще одна популярная концепция, поддерживаемая пакетом `java.time`, — это *временная метка* (*timestamp*). В любой ситуации, когда надо отслеживать передачу информации, бывает полезно зарегистрировать точное время появления или изменения информации. Для хранения этих моментов времени иногда используется класс `java.util.Date`, но удобнее класс `java.time.Instant`, который содержит все необходимое для работы с временными метками и обладает всеми преимуществами других классов пакета `java.time`:

```
jshell> Instant time1 = Instant.now()
time1 ==> 2019-12-14T15:38:29.033954Z

jshell> Instant time2 = Instant.now()
time2 ==> 2019-12-14T15:38:46.095633Z

jshell> time1.isAfter(time2)
$54 ==> false

jshell> time1.plus(3, ChronoUnit.DAYS)
$55 ==> 2019-12-17T15:38:29.033954Z
```

Если вы работаете с датами и временем, то пакет `java.time` станет ценным дополнением к Java. Он предоставляет в ваше распоряжение полноценный и хорошо спроектированный инструментарий для обработки этих данных — и никакие сторонние библиотеки вам не понадобятся!

Другие полезные средства

Мы рассмотрели некоторые структурные элементы Java, включая строки и числа, а также даты (одни из самых распространенных комбинаций строк и чисел) в классах `LocalDate` и `LocalTime`. Надеемся, что эта глава дала вам некоторое представление о том, как Java работает со многими типичными элементами, необходимыми для решения реальных задач. Обязательно ознакомьтесь с документацией пакетов `java.util`, `java.text` и `java.time` — в ней описаны вспомогательные инструменты, которые могут вам пригодиться в работе. Например, загляните в документацию класса `java.util.Random`, с помощью которого можно генерировать случайные координаты деревьев, изображенных на рис. 8.1. Важно отметить, что иногда «вспомогательные» задачи оказываются достаточно сложными и требуют особого внимания к под-

робностям. Вы найдете в интернете много примеров кода и даже полноценные библиотеки, написанные другими разработчиками; они помогут вам быстрее справиться с делом.

На следующем этапе мы начнем развивать эти фундаментальные концепции. Популярность языка Java объясняется тем, что наряду со своими основными инструментами он поддерживает и более мощные средства программирования. Одним из таких средств, с самого начала сыгравшим важную роль в успехе Java, стала встроенная поддержка многопоточности. Потoki позволяют программисту эффективнее использовать мощные современные компьютеры и писать такие приложения, которые «не тормозят» даже при обработке множества сложных задач. Итак, давайте разберемся, как пользоваться этим фирменным средством Java.

ГЛАВА 9

Потоки

Мы воспринимаем как само собой разумеющееся, что современные компьютеры умеют справляться со множеством приложений сразу, а параллельно с приложениями выполняются задачи операционной системы, — и нам кажется, что все это работает одновременно. Почти в каждом современном компьютере есть многоядерный процессор (один или даже несколько), способный достичь впечатляющей степени параллелизма. При этом операционная система, как и прежде, «жонглирует» приложениями на своем, более высоком уровне, переключая их настолько быстро, что возникает иллюзия одновременного выполнения.

Давным-давно единицей параллельного выполнения в таких системах был *процесс*. Для операционной системы процесс напоминал «черный ящик», который сам решал, что ему делать. Если приложению требовалась более высокая степень параллелизма, оно могло достичь этого только запуском нескольких процессов и организацией взаимодействия между ними. Это решение было тяжеловесным и не отличалось элегантностью. Позднее появилась концепция *потоков (threads)*¹. Потоки обеспечивают более структурированный параллелизм внутри процесса под контролем самого приложения. Потоки придумали уже давно, но поначалу работать с ними было довольно сложно. В Java поддержка потоков встроена в язык, что очень упростило работу с ними. Средства параллелизма Java поддерживают распространенные паттерны и приемы многопоточных приложений, поднимая их до уровня действующих Java API. В совокупности это означает, что в языке Java многопоточность поддерживается как на платформенном, так и на высоком уровне. Также это означает, что Java API обладают всеми преимуществами многопоточности. Поэтому важно, чтобы вы хотя бы в общих чертах освоили эти концепции при знакомстве с Java. Не каждому разработчику приходится явно использовать параллелизм и потоки, но зависящие от них инструменты языка нужны в повседневной работе практически всем.

¹ Речь идет о потоках выполнения (threads). В главе 11 мы рассмотрим потоки данных (streams).

Потоки являются неотъемлемой частью многих Java API, особенно относящихся к приложениям на стороне клиента, к графике и звуку. Например, в компонентах Swing метод `paint()` вызывается не напрямую из приложения, а из отдельного потока в исполнительной системе Java. В любой момент времени множество таких потоков может работать параллельно с вашим приложением. Потоки Java также присутствуют на стороне сервера, обслуживая все запросы и выполняя компоненты вашего приложения. Важно понимать, какое место занимает в этой среде ваш код.

В этой главе рассматривается создание приложений, которые явно создают свои потоки и пользуются ими. Сначала мы поговорим о низкоуровневой поддержке потоков, встроенной в язык Java, а затем подробно обсудим пакет средств работы с потоками `java.util.concurrent`.

Знакомство с потоками

На концептуальном уровне *поток* (*thread*) представляет собой отдельную последовательность выполнения команд в программе. Поток напоминает более знакомые нам процессы, не считая того, что потоки одного приложения связаны намного ближе и совместно используют одно состояние. Происходящее можно сравнить с полем для гольфа, на котором несколько гольфистов могут находиться одновременно. Потоки совместно используют одну рабочую область. Для них доступны одни и те же объекты внутри своего приложения, включая статические переменные и переменные экземпляров. Однако потоки содержат собственные копии локальных переменных, подобно тому как игроки находятся на одном поле для гольфа, но не используют одни и те же принадлежности (например, клюшки и мячи).

У нескольких потоков в приложении могут возникнуть те же проблемы, что и у гольфистов, — плохая синхронизация. Несколько групп игроков не могут отыгрывать одну дорожку одновременно; точно так же несколько потоков не смогут обратиться к одной переменной без соответствующей синхронизации. У некоторых из них неизбежно возникнут проблемы. Поток может зарезервировать право использования объекта до того, как завершит с ним работу, — по аналогии с тем, как группа гольфистов получает монопольные права доступа на дорожку, пока она не будет пройдена. А более важный поток может поднять свой приоритет, тем самым заявляя свое преимущественное право на работу с каким-либо ресурсом.

Конечно, дьявол кроется в деталях, и эти детали издавна усложняли использование потоков. К счастью, Java упрощает задачи создания потоков, управления ими и их координации благодаря интеграции в язык некоторых важных концепций.

Скорее всего, при первой попытке работы с потоками у вас возникнут трудности, потому что для создания потока нужны почти все ваши новые знания о Java. Чтобы избежать недоразумений, запомните, что в выполнении потока задействованы две стороны: объект языка `Java Thread`, представляющий сам поток, и произвольный целевой объект с методом, который должен выполняться потоком. Скоро вы увидите, что ценой некоторых ухищрений эти две роли можно объединить, но этот особый случай изменяет лишь упаковку, но не сами отношения.

Класс `Thread` и интерфейс `Runnable`

Выполняемый код в Java всегда связан с объектом `Thread`, начиная с главного метода, который запускается виртуальной машиной Java для старта вашего приложения. Новый поток порождается при создании экземпляра класса `java.lang.Thread`. Объект `Thread` представляет реальный поток в интерпретаторе Java и служит своего рода дескриптором для управления им и для координации его выполнения. С его помощью вы можете запустить поток, дождаться завершения потока, заставить его приостановиться на некоторое время или прервать его выполнение. Конструктор класса `Thread` получает информацию о том, где поток должен начать свое выполнение. На концептуальном уровне нам хотелось бы просто сообщить конструктору, какой метод должен выполняться. Это можно сделать несколькими способами; Java 8 поддерживает ссылки на методы, которые позволяют это сделать. Но сейчас мы выберем обходной путь и воспользуемся интерфейсом `java.lang.Runnable` для создания или пометки объекта, содержащего «исполняемый» метод. Интерфейс `Runnable` определяет один метод общего назначения `run()`:

```
public interface Runnable {  
    abstract public void run();  
}
```

Каждый поток начинает свое существование с выполнения метода `run()` в объекте `Runnable` — «целевом объекте», переданном конструктору потока. Метод `run()` может содержать любой код, но этот метод должен быть открытым, не получать аргументов, не иметь возвращаемого значения и не выдавать проверяемых исключений.

Любой класс, содержащий подходящий метод `run()`, может объявить, что он реализует интерфейс `Runnable`. Тогда экземпляр этого класса становится исполняемым методом, который может служить целью нового потока. Если вы не хотите включать метод `run()` прямо в объект (а очень часто это бывает нежелательно), вы всегда можете создать класс-адаптер, который сыграет роль `Runnable`. В этом случае метод `run()` адаптера может вызвать любой нужный метод после запуска потока. Примеры таких решений будут приведены далее.

Создание и запуск потоков

Только что созданный поток бездействует, пока вы не приведете его в действие вызовом метода `start()`. Тогда поток «просыпается» и переходит к выполнению метода `run()` целевого объекта. Метод `start()` может быть вызван только один раз за жизненный цикл потока. После того как поток будет запущен, он продолжает выполняться, пока метод `run()` целевого объекта не вернет управление (или пока не выдаст непроверяемое исключение). У метода `start()` есть парный метод `stop()`, который безвозвратно уничтожает поток. Тем не менее этот метод считается устаревшим и пользоваться им не рекомендуется. Вскоре мы объясним причину и приведем примеры более корректной остановки потоков. Также мы рассмотрим некоторые методы, которые могут использоваться для контроля за выполнением потока.

Рассмотрим пример. В классе `Animator` реализуется метод `run()`, управляющий циклом прорисовки, который служит в нашей игре для обновления `Field`:

```
class Animator implements Runnable {
    boolean animating = true;

    public void run() {
        while ( animating ) {
            // Переместить яблоки на один "кадр"
            // Перерисовать поле
            // Сделать паузу
            ...
        }
    }
}
```

Чтобы его использовать, мы создаем объект `Thread`, передаем ему экземпляр `Animator` как целевой объект и вызываем его метод `start()`. Все эти действия можно выполнить явно:

```
Animator myAnimator = new Animator();
Thread myThread = new Thread(myAnimator);
myThread.start();
```

Здесь создается экземпляр класса `Animator`, который передается в аргументе конструктора в `myThread`. Как показано на рис. 9.1, при вызове метода `start()` объект `myThread` начинает выполнять метод `run()` класса `Animator`. Вперед!

Естественное создание потока

Интерфейс `Runnable` позволяет создать произвольный объект и назначить его целевым для потока, как было сделано в предыдущем примере. Это самый важный случай общего применения класса `Thread`. В большинстве ситуаций, в которых

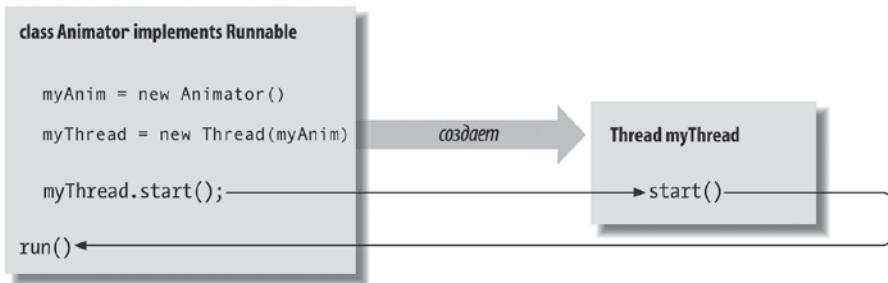


Рис. 9.1. Animator как реализация Runnable

возникает необходимость использования потоков, создается класс (возможно, простой класс-адаптер), реализующий интерфейс `Runnable`.

Тем не менее надо представить и другой способ создания потоков. В этом варианте целевой класс определяется как субкласс типа, который уже поддерживает `Runnable`. Оказывается, сам класс `Thread` для удобства реализует интерфейс `Runnable`; он содержит метод `run()`, который можно напрямую переопределить в соответствии с вашими целями:

```

class Animator extends Thread {
    boolean animating = true;

    public void run() {
        while ( animating ) {
            // Рисование в окне
            ...
        }
    }
}
  
```

Основа класса `Animator` выглядит примерно так же, как прежде, за исключением того, что наш субкласс теперь стал субклассом `Thread`. В соответствии с этой схемой конструктор по умолчанию класса `Thread` назначает самого себя целью, то есть по умолчанию `Thread` выполняет собственный метод `run()` при вызове метода `start()`, как показано на рис. 9.2. Теперь наш субкласс может просто переопределить метод `run()` из класса `Thread`. (Сам класс `Thread` просто определяет пустой метод `run()`.)

Затем мы создаем экземпляр класса `Animator` и вызываем его метод `start()` (который также наследуется от `Thread`):

```

Animator bouncy = new Animator();
bouncy.start();
  
```

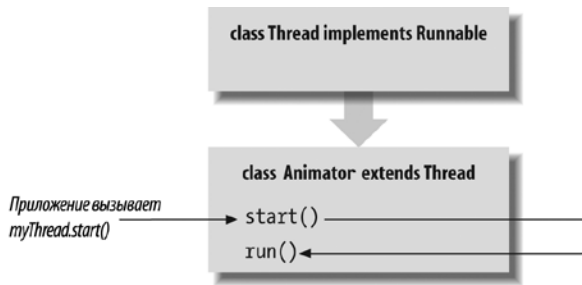


Рис. 9.2. Animator как субкласс Thread

Также можно приказать объекту `Animator` запускать его поток при создании:

```
class Animator extends Thread {  
  
    Animator () {  
        start();  
    }  
    ...  
}
```

Здесь объект `Animator` просто вызывает свой метод `start()` при создании экземпляра. (Вероятно, лучше запускать и останавливать объекты после их создания, вместо того чтобы полагаться на запуск потоков как на скрытый побочный эффект создания объекта, но для примера сойдет.)

Может показаться, что субклассирование `Thread` позволяет удобно упаковать воедино поток и его целевой метод `run()`. Тем не менее это решение не всегда оказывается лучшим с точки зрения проектирования. Если вы субклассируете `Thread` для реализации потока, вы тем самым заявляете, что вам нужен новый тип объекта, являющийся разновидностью `Thread`, который предоставляет весь открытый API класса `Thread`. Заманчиво выглядит идея взять объект, основной задачей которого является выполнение задачи, и превратить его в `Thread`, но реальные ситуации, в которых вы хотите создать субкласс `Thread`, встречаются не очень часто. Обычно бывает более естественно формировать структуру классов на основании требований вашей программы и использовать `Runnable`, чтобы связать выполнение с логикой вашей программы.

Управление потоками

Мы уже показали, как использовать метод `start()` для запуска выполнения нового потока. Существует ряд других методов экземпляров для явного управления выполнением потока.

- Статический метод `Thread.sleep()` приостанавливает на заданный период времени (с некоторой точностью) поток, выполняемый в настоящий момент; при этом не расходуется (или почти не расходуется) процессорное время.
- Методы `wait()` и `join()` координируют выполнение двух и более потоков. Они будут подробно рассмотрены при обсуждении синхронизации потоков далее в этой главе.
- Метод `interrupt()` возобновляет поток, приостановленный операцией `sleep()` или `wait()` либо заблокированный иным способом в продолжительной операции ввода-вывода¹.

Устаревшие методы

Также необходимо упомянуть о трех устаревших методах управления потоками: `stop()`, `suspend()` и `resume()`. Метод `stop()` является парным по отношению к `start()`; он уничтожает поток. Методы `start()` и `stop()` могут вызываться только один раз за весь жизненный цикл потока. С другой стороны, устаревшие методы `suspend()` и `resume()` использовались для произвольной приостановки и последующего перезапуска потока.

И хотя эти устаревшие методы все еще поддерживаются в Java (и скорее всего, будут поддерживаться всегда), их не надо использовать при разработке нового кода. Проблема `stop()` и `suspend()` заключается в том, что эти методы перехватывают управление над выполнением потоков жестко и некоординированно. Это усложняет программирование; приложение не всегда может предвидеть возможное прерывание его выполнения в любой точке и корректно восстановиться после него. Более того, когда управление потоком перехватывается одним из этих способов, исполнительная система Java должна освободить все свои внутренние блокировки, использованные для синхронизации потоков. Это может привести к неожиданному поведению, а в случае вызова метода `suspend()`, который не освобождает эти блокировки, легко может привести к взаимной блокировке.

Лучший способ управления выполнением потока (требующий чуть больше работы с вашей стороны) основан на включении в код потока простой логики, использующей переменные-мониторы (если эти переменные относятся к логическому типу, они иногда называются «флагами») — возможно, в сочетании с методом `interrupt()`. Эта логика позволяет «разбудить» приостановленный поток. Иначе говоря, чтобы заставить поток прервать или продолжить выполнение того, чем он занимается в настоящее время, следует вежливо попросить, вместо того чтобы неожиданно выдергивать ковер у него из-под ног. В примерах этой книги так или иначе используется именно этот способ.

¹ Метод `interrupt()` работает нестабильно во всех реализациях Java, особенно в ранних.

Метод `sleep()`

Часто требуется приказать потоку прервать выполнение, или «заснуть», на фиксированный промежуток времени. Пока поток приостановлен или иным способом блокируется от получения ввода, он не потребляет процессорное время и не конкурирует с другими потоками за вычислительные ресурсы. Для этого можно вызвать статический метод `Thread.sleep()`, который влияет на текущий выполняемый поток. Вызов заставляет поток перейти в состояние бездействия на заданное количество миллисекунд:

```
try {  
    // Текущий поток  
    Thread.sleep( 1000 );  
} catch ( InterruptedException e ) {  
    // Кто-то преждевременно нас разбудил  
}
```

Метод `sleep()` может выдать исключение `InterruptedException`, если он будет прерван другим потоком в результате вызова метода `interrupt()` (см. далее). Как было показано выше, поток может перехватить это исключение и воспользоваться возможностью для выполнения некоторых действий (например, проверить переменную, чтобы выяснить, надо ли завершаться) или же выполнить некоторую подготовку, а затем перейти в состояние приостановки.

Метод `join()`

Наконец, если операция должна координироваться с другим потоком (а именно дожидаться, пока он завершит свою работу), вы можете использовать метод `join()`. Вызов метода `join()` заставляет вызывающую сторону заблокироваться до завершения целевого потока. Также можно вызвать `join()` с передачей продолжительности ожидания в миллисекундах. Это очень примитивная форма синхронизации потока. В Java есть более универсальные и мощные механизмы координации активности потоков, включая методы `wait()` и `notify()`, а также высокоуровневые API из пакета `java.util.concurrent`. Эти темы вам придется изучать в основном самостоятельно, но отметим, что язык Java упростил написание многопоточных приложений по сравнению со многими своими предшественниками.

Метод `interrupt()`

Метод `interrupt()` упоминался как средство возобновления работы потока, бездействующего из-за вызова `sleep()`, `wait()` или долгой операции ввода-вывода. Любой поток, не работающий постоянно (не находящийся в «жестком цикле»), должен периодически входить в одно из этих состояний, поэтому в какой-то точке кода поток должен получить приказ остановиться. При прерывании потока устанавливается его флаг *статуса прерывания* (*interrupt status*). Это может

произойти в любой момент, независимо от того, бездействует поток или нет. Поток может проверить это состояние методом `isInterrupted()`. Другая форма метода — `isInterrupted(boolean)` — получает логическое значение, которое указывает, надо ли сбросить статус прерывания. Таким образом, поток может использовать статус прерывания как флаг и как сигнал.

Во всяком случае, предполагаемая функциональность метода именно такова. Но разработчикам Java с самого начала не удавалось добиться его корректной работы во всех возможных ситуациях. В первых виртуальных машинах Java (до версии 1.1) метод `interrupt()` вообще не работал. И даже в более новых версиях все еще есть проблемы с прерыванием вызовов ввода-вывода. Под «вызовом ввода-вывода» имеется в виду, что приложение блокируется в методе `read()` или `write()` на время перемещения байтов из источника (из файла, из сети) или в источник. В таком случае при выполнении `interrupt()` должно выдаваться исключение `InterruptedException`. Но этот механизм не всегда работает надежно. В свое время решения этой проблемы ожидали от нового фреймворка ввода-вывода `java.nio`, представленного в Java 1.4. При прерывании потока, связанного с операцией NIO, он «просыпается» и при этом автоматически закрывается поток ввода-вывода (I/O stream), называемый «каналом». (За дополнительной информацией о пакете NIO обращайтесь к главе 11.)

Снова об анимации

Как упоминалось в начале главы, одной из стандартных задач при разработке графических интерфейсов становится управление анимациями. Иногда анимации — это лишь маленькие украшения интерфейса, а в других случаях они занимают центральное место в приложении — как в нашей игре с бросанием яблок. Известны разные способы реализации анимаций. Мы рассмотрим использование простых потоков, метода `sleep()` и таймера в сочетании с покадровой прорисовкой фаз движения. Это популярный способ, простой для понимания, и мы продемонстрируем его при анимации летящих яблок в нашей игре.

Мы можем использовать поток (похожий на тот, о котором шла речь в разделе «Создание и запуск потоков», с. 305) для создания неплохой анимации. Основная идея заключается в том, чтобы нарисовать в нужных местах все анимируемые объекты, сделать паузу и переместить их на новые места, а затем все повторить. Для начала посмотрим, как нарисовать некоторые объекты игрового поля без анимации.

```
// Из класса Field...
protected void paintComponent(Graphics g) {
    g.setColor(fieldColor);
    g.fillRect(0,0, getWidth(), getHeight());
    physicist.draw(g);
    for (Tree t : trees) {
        t.draw(g);
    }
}
```

```
    }  
    for (Apple a : apples) {  
        a.draw(g);  
    }  
}  
  
// Из класса Apple...  
public void draw(Graphics g) {  
    // Выбрать красный цвет для яблока, а затем нарисовать его!  
    g.setColor(Color.RED);  
    g.fillOval(x, y, scaledLength, scaledLength);  
}
```

Все просто. Сначала рисуется фоновое поле, потом физик, затем деревья и, наконец, яблоки. Такой порядок прорисовки гарантирует, что яблоки будут выводиться «поверх» всех остальных элементов. Класс `Field` переопределяет метод среднего уровня `paintComponent()`, доступный во всех графических элементах `Swing` для произвольной прорисовки, но эта тема будет рассматриваться в главе 10.

Давайте задумаемся над тем, что же изменяется на экране в ходе игры. В действительности имеются два «движущихся» элемента: яблоко, которым физик прицеливается с верха своей башни, и яблоки, летящие после броска. Мы знаем, что «анимация» прицеливания всего лишь реагирует на обновление фигуры физика при перемещении ползунка. Отдельной анимации тут не требуется. Таким образом, нам необходимо сосредоточиться только на обработке летящих яблок. Функция «следующего кадра» должна перемещать каждое активное яблоко под воздействием гравитации. Эта работа обеспечивается двумя методами. Исходные условия настраиваются в методе `toss()` в соответствии со значениями ползунков прицеливания и силы броска, а в методе `step()` выполняется перемещение яблока.

```
// Из класса Apple...  
  
public void toss(float angle, float velocity) {  
    lastStep = System.currentTimeMillis();  
    double radians = angle / 180 * Math.PI;  
    velocityX = (float)(velocity * Math.cos(radians) / mass);  
    // Начинаем с отрицательного значения velocity,  
    // так как направлению вверх соответствуют меньшие значения y  
    velocityY = (float)(-velocity * Math.sin(radians) / mass);  
}  
  
public void step() {  
    // Проверяем, что перемещение вообще происходит; переменная  
    // lastStep используется в качестве сторожевого значения  
    if (lastStep > 0) {  
        // Применяем силу гравитации  
        long now = System.currentTimeMillis();
```

```

        float slice = (now - lastStep) / 1000.0f;
        velocityY = velocityY + (slice * Field.GRAVITY);
        int newX = (int)(centerX + velocityX);
        int newY = (int)(centerY + velocityY);
        setPosition(newX, newY);
    }
}

```

Итак, методы обновления яблок готовы. Их можно заключить в цикл анимации, который выполняет вычисления по обновлению, перерисовывает поле, делает паузу и повторяется.

```

public static final int STEP = 40; // Продолжительность кадра анимации
                                   // в миллисекундах

// ...

class Animator implements Runnable {
    public void run() {
        // "animating" - это глобальная переменная, которая позволяет
        // остановить анимацию и сэкономить ресурсы при отсутствии
        // активных яблок, которые нужно перемещать
        while (animating) {
            System.out.println("Stepping " + apples.size() + " apples");
            for (Apple a : apples) {
                a.step();
                detectCollisions(a);
            }
            Field.this.repaint();
            cullFallenApples();
            try {
                Thread.sleep((int)(STEP * 1000));
            } catch (InterruptedException ie) {
                System.err.println("Animation interrupted");
                animating = false;
            }
        }
    }
}

```

Мы используем эту реализацию `Runnable` в простом потоке. Наш класс `Field` будет хранить экземпляр потока, и в него включен следующий простой метод запуска анимации:

```

Thread animationThread;

// ...

void startAnimation() {
    animationThread = new Thread(new Animator());
    animationThread.start();
}

```


При помощи событий пользовательского интерфейса, которые будут рассматриваться в разделе «События», с. 365, мы будем запускать яблоки по команде. Пока первое яблоко просто запускается в самом начале игры. Мы знаем, что скриншот на рис. 9.3 не очень впечатляет, но поверьте: в движении все выглядит просто потрясающе! :)

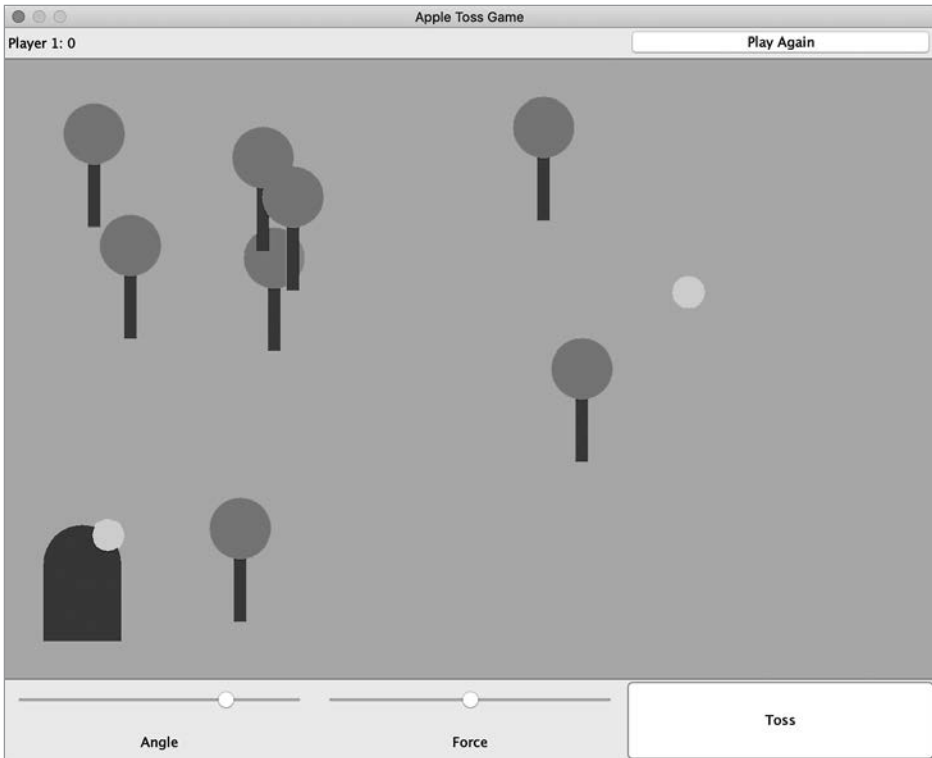


Рис. 9.3. Бросание яблок в действии

Смерть потоков

Поток продолжает выполняться до наступления одного из следующих событий:

- Он явно возвращает управление из своего целевого метода `run()`.
- Он сталкивается с неперехваченным исключением времени выполнения.
- Вызывается нехороший и устаревший метод `stop()`.

Что произойдет, если ни одно из этих событий не произойдет, а метод `run()` потока никогда не завершится? Такой поток будет продолжать свое суще-

ствование даже после того, как создавшая его часть приложения завершится. А значит, мы должны следить за тем, чтобы все потоки в конечном итоге завершились. В ином случае в приложении могут остаться никому не принадлежащие потоки, которые будут без надобности поглощать ресурсы компьютера или поддерживать существование приложения, когда оно уже должно было завершиться.

Во многих случаях мы хотим создать фоновые потоки для выполнения простых периодических задач в приложении. При помощи метода `setDaemon()` можно пометить поток как поток-демон. Такой поток должен быть уничтожен, когда у приложения не останется ни одного потока, который бы не был демоном. Обычно интерпретатор Java продолжает работать до тех пор, пока все потоки не завершатся. Но когда остаются только потоки-демоны, интерпретатор завершается.

Пример использования потоков-демонов:

```
class Devil extends Thread {
    Devil() {
        setDaemon( true );
        start();
    }
    public void run() {
        // Разные демонические дела
    }
}
```

В этом примере поток `Devil` устанавливает свой статус демона при создании. Если в тот момент, когда приложение в целом завершено, остаются какие-либо потоки `Devil`, исполнительная система уничтожит их за нас. Нам не нужно беспокоиться об их завершении.

Демоны в основном используются в автономных приложениях Java и в реализациях серверных фреймворков, но не в компонентных приложениях (в которых меньшие фрагменты кода выполняются внутри большего). Так как компоненты выполняются внутри другого приложения Java, все созданные ими потоки-демоны могут продолжить свое существование до выхода из управляющего приложения, — скорее всего, это не то, что вам нужно. В таких приложениях можно воспользоваться объектами `ThreadGroup`, которые содержат все потоки, создаваемые подсистемами или компонентами, а затем освобождают их при необходимости.

И последнее замечание по поводу корректного уничтожения потоков. Очень многие разработчики в своем первом приложении, использующем компонент `Swing`, сталкиваются с типичной проблемой: их приложение не завершается, а виртуальная машина Java продолжает «висеть», когда все уже завершено.

Java создает при работе с графикой GUI-поток (поток пользовательского интерфейса) для обработки событий ввода и прорисовки. GUI-поток не является демоном, поэтому он не завершается автоматически при завершении всех остальных потоков приложения и разработчик должен вызвать `System.exit()` явно. (И если подумать, это вполне логично, так как большинство GUI-приложений управляется событиями и просто ожидает ввода от пользователя, в ином случае они бы просто завершались после завершения их стартового кода.)

Синхронизация

Каждый поток «мыслит» самостоятельно. Обычно поток занимается своим делом, не обращая внимания на то, чем заняты другие потоки приложения. Потоки могут квантоваться по времени; это означает, что операционная система по очереди выделяет им произвольные промежутки времени для выполнения. В многопроцессорных (многоядерных) системах несколько разных потоков даже могут выполняться одновременно на разных процессорах (ядрах). Этот раздел посвящен координации активности двух и более потоков, чтобы они могли работать совместно и не конфликтовали из-за использования одних и тех же переменных и методов.

Java предоставляет несколько простых структур для синхронизации активности потоков. Все они основаны на концепции *мониторов* — это часто применяемая схема синхронизации. Вам не обязательно глубоко понимать принципы работы мониторов, чтобы пользоваться ими, но желательно хотя бы иметь общее представление.

По своему принципу работы монитор напоминает замок (*блокировку* в терминологии синхронизации). Этот замок прикрепляется к ресурсу, к которому могут обращаться несколько потоков; но в любой момент времени с ресурсом должен работать только один поток. Представьте туалет с дверным замком; пока он открыт, вы можете зайти и запереть дверь. Пока ресурс не используется, любой поток может получить доступ к замку и работать с ресурсом. Завершив работу, поток освобождает замок — вы отпираете дверь и оставляете ее открытой для следующего посетителя. Но если другой поток уже «запер дверь» и работает с ресурсом, то всем остальным потокам придется ждать, пока текущий поток завершит работу с ресурсом и покинет его. Все выглядит так, словно туалет занят на момент вашего прихода: вам придется дожидаться, пока текущий поток завершит работу и откроет дверь.

К счастью, в Java процесс синхронизации доступа к ресурсам устроен совсем просто. Java берет на себя захват и освобождение блокировок («замков»). Все, что вам надо сделать, — указать ресурсы, требующие синхронизации.

Организация последовательного доступа к методам

Самой распространенной задачей, требующей синхронизации потоков Java, является организация последовательного доступа к ресурсу (объекту), иначе говоря, принятие мер к тому, чтобы в любой момент времени с объектом или переменной мог работать только один поток. В Java с каждым объектом связывается *блокировка*. Если говорить конкретнее, собственной блокировкой обладает каждый класс и каждый экземпляр класса. Ключевое слово `synchronized` помечает те места, в которых поток должен захватить блокировку перед продолжением работы.

Допустим, вы реализовали класс `SpeechSynthesizer`, который содержит метод `say()`. Вы не хотите, чтобы сразу несколько потоков могли выполнять `say()` одновременно, поэтому метод `say()` помечается ключевым словом `synchronized`. Оно означает, что поток должен захватить блокировку объекта `SpeechSynthesizer` перед выполнением `say()`:

```
class SpeechSynthesizer {  
    synchronized void say( String words ) {  
        // ...  
    }  
}
```

Так как `say()` является методом экземпляра, поток должен захватить блокировку используемого экземпляра `SpeechSynthesizer`, прежде чем он сможет активизировать метод `say()`. После завершения `say()` он уступает блокировку, что позволяет следующему ожидающему потоку захватить блокировку и выполнить метод.

При этом неважно, принадлежит ли поток самому экземпляру `SpeechSynthesizer` или какому-то другому объекту; каждый поток должен захватывать одну и ту же блокировку, принадлежащую экземпляру `SpeechSynthesizer`. Если бы метод `say()` был методом класса (статическим методом), а не методом экземпляра, его тоже можно было бы пометить ключевым словом `synchronized`. В таком случае, когда экземпляр не участвует, блокировка принадлежит самому классу.

Часто требуется синхронизировать несколько методов одного класса, чтобы в любой момент времени только один метод изменял или просматривал части класса. Все статические `synchronized`-методы класса используют одну блокировку, связанную с объектом самого класса. Аналогичным образом все методы экземпляра в классе используют одну блокировку экземпляра. Таким образом Java может гарантировать, что в любой момент выполняется только один набор `synchronized`-методов. Например, класс `SpreadSheet` может содержать группу переменных экземпляра, представляющих значения ячеек, а также методы, оперирующие с ячейками строки:

```

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()` обращаются к значениям ячеек. Мы видим, что при изменении значений переменных в `setRow()` могут возникнуть проблемы: один поток может изменить значения переменных в `setRow()` в тот момент, когда другой поток читает значения в `sumRow()`. Чтобы этого не произошло, оба метода помечены ключевым словом `synchronized`. В любой момент времени может выполняться только один из синхронизированных потоков. Если поток находится на середине выполнения `setRow()`, когда другой поток вызывает `sumRow()`, то второй поток ожидает, пока первый поток завершит выполнение `setRow()`, и только после этого выполняет `sumRow()`. Эта синхронизация позволяет защитить целостность данных `SpreadSheet`. А самое замечательное, что всю работу с захватами блокировок и ожиданиями делает Java; для программиста они прозрачны.

Кроме синхронизации целых методов, ключевое слово `synchronized` может использоваться в специальной конструкции для защиты произвольных блоков кода. В этой форме оно также получает аргумент, определяющий объект, для которого должна быть захвачена блокировка:

```

synchronized ( myObject ) {
    // Функциональность, требующая монопольного доступа к ресурсам
}

```

Этот блок может размещаться в любом методе. При его достижении поток должен захватить блокировку для `myObject` перед продолжением работы. Таким образом мы можем синхронизировать методы (или части методов) в разных классах по тому же принципу, что и методы одного класса.

Следовательно, метод экземпляра с пометкой `synchronized` эквивалентен методу, команды которого синхронизируются по текущему объекту. Взгляните на фрагмент:

```

synchronized void myMethod () {
    ...
}

```

Он эквивалентен следующему фрагменту:

```
void myMethod () {
    synchronized ( this ) {
        ...
    }
}
```

Основы синхронизации можно продемонстрировать на классическом примере «производитель и потребитель». Имеются общие ресурсы; производители создают новые ресурсы, а потребители берут и используют эти ресурсы. Примером может послужить группа веб-роботов, обрабатывающих изображения в интернете. «Производителем» в такой ситуации может быть поток (или несколько потоков), выполняющий фактическую работу по загрузке и разбору веб-страниц в поисках изображений и их URL-адресов. Найденные URL-адреса помещаются в общую очередь, а поток-«потребитель» (или потоки) выбирает следующий URL в очереди и непосредственно загружает изображение в файловую систему или в базу данных. Мы не будем выполнять здесь реальный ввод-вывод (файлы и сети подробнее рассматриваются в главе 11), а ограничимся созданием нескольких потоков-производителей и потоков-потребителей, чтобы продемонстрировать работу синхронизации.

Синхронизация очереди URL-адресов

Начнем с очереди, в которой будут храниться URL-адреса. Никакие хитроумные структуры данных для очереди не потребуются; это обычный список, в котором URL-адреса (в виде `String`) добавляются в конец и извлекаются с начала. Для реализации очереди будет использоваться список `LinkedList`, похожий на список `ArrayList` из главы 7. Эта структура спроектирована для эффективного выполнения обращений и тех операций, которые понадобятся нам при работе с очередью.

```
package ch09;

import java.util.LinkedList;

public class URLQueue {
    LinkedList<String> urlQueue = new LinkedList<>();

    public synchronized void addURL(String url) {
        urlQueue.add(url);
    }

    public synchronized String getURL() {
        if (!urlQueue.isEmpty()) {
            return urlQueue.removeFirst();
        }
        return null;
    }
}
```

```

    public boolean isEmpty() {
        return urlQueue.isEmpty();
    }
}

```

Заметьте, что не все методы имеют модификатор **synchronized**! Мы разрешаем любому методу спросить, пуста очередь или нет, без приостановки других потоков, которые могут выполнять добавление или удаление. Это означает, что при проверке может быть получен неправильный ответ (если другие потоки сработают в неудачном сочетании), но наша система отчасти отказоустойчива, поэтому эффективность решения без блокировки в данном случае (только для проверки размера очереди) важнее, чем идеальная точность ответа¹.

Теперь, когда мы знаем, как хранить и читать URL-адреса, можно переходить к созданию классов производителя и потребителя. Производитель запускает простой цикл, который генерирует фиктивные URL-адреса, снабжает их префиксом с идентификатором производителя и сохраняет в очереди. Метод `run()` для класса производителя выглядит так:

```

public void run() {
    for (int i = 1; i <= urlCount; i++) {
        String url = "https://some.url/at/path/" + i;
        queue.addURL(producerID + " " + url);
        System.out.println(producerID + " produced " + url);
        try {
            Thread.sleep(delay.nextInt(500));
        } catch (InterruptedException ie) {
            System.err.println("Producer " + producerID +
                               " interrupted. Quitting.");
            break;
        }
    }
}

```

Класс потребителя в общем-то похож на него, не считая очевидного исключения с извлечением URL-адресов из очереди. Он извлекает URL, снабжает его префиксом с идентификатором потребителя и начинает все заново, пока производители не перестанут производить, а очередь не опустеет.

```

public void run() {
    while (keepWorking || !queue.isEmpty()) {
        String url = queue.getURL();
        if (url != null) {

```

¹ Несмотря на отказоустойчивость, в современных многоядерных системах без идеально точной информации может возникнуть хаос. А достичь идеала непросто! Если вы собираетесь работать с потоками в реальных условиях, обязательно прочитайте книгу Брайана Гетца «Java Concurrency на практике» (СПб.: Питер).

```

        System.out.println(consumerID + " consumed " + url);
    } else {
        System.out.println(consumerID + " skipped empty queue");
    }
    try {
        Thread.sleep(delay.nextInt(1000));
    } catch (InterruptedException ie) {
        System.err.println("Consumer " + consumerID + " interrupted.
            Quitting.");
        break;
    }
}
}
}

```

Моделирование можно начать с малых чисел. Вот пример с двумя производителями и двумя потребителями, при этом каждый производитель создает всего три URL:

```

package ch09;

public class URLEDemo {
    public static void main(String args[]) {
        URLQueue queue = new URLQueue();
        URLProducer p1 = new URLProducer("P1", 3, queue);
        URLProducer p2 = new URLProducer("P2", 3, queue);
        URLConsumer c1 = new URLConsumer("C1", queue);
        URLConsumer c2 = new URLConsumer("C2", queue);
        System.out.println("Starting...");
        p1.start();
        p2.start();
        c1.start();
        c2.start();
        try {
            // Ожидать, пока производители перестанут создавать URL-адреса
            p1.join();
            p2.join();
        } catch (InterruptedException ie) {
            System.err.println("Interrupted waiting for producers to finish");
        }
        c1.setKeepWorking(false);
        c2.setKeepWorking(false);
        try {
            // Ожидать, пока потребители освободят очередь
            c1.join();
            c2.join();
        } catch (InterruptedException ie) {
            System.err.println("Interrupted waiting for consumers to finish");
        }
        System.out.println("Done");
    }
}

```


Но даже с таким небольшим количеством потоков мы видим эффект от многопоточности:

```
Starting...
C1 skipped empty queue
C2 skipped empty queue
P2 produced https://some.url/at/path/1
P1 produced https://some.url/at/path/1
P1 produced https://some.url/at/path/2
P2 produced https://some.url/at/path/2
C2 consumed P2 https://some.url/at/path/1
P2 produced https://some.url/at/path/3
P1 produced https://some.url/at/path/3
C1 consumed P1 https://some.url/at/path/1
C1 consumed P1 https://some.url/at/path/2
C2 consumed P2 https://some.url/at/path/2
C1 consumed P2 https://some.url/at/path/3
C1 consumed P1 https://some.url/at/path/3
Done
```

Обратите внимание: идеального чередования потоков нет, но каждый поток получает по крайней мере какое-то рабочее время. И мы видим, что потребители не привязываются к конкретным производителям. Снова идея заключается в том, чтобы эффективно использовать ограниченные ресурсы. Производители могут продолжать добавлять задачи, не беспокоясь о том, сколько времени займет выполнение каждой задачи или кому она будет поручена. В свою очередь, потребители могут захватывать задачи, не беспокоясь о других потребителях. Если потребитель получает простую задачу и завершает ее ранее других потребителей, то он может вернуться и немедленно получить новую задачу.

Попробуйте выполнить этот код самостоятельно и увеличить некоторые числа. Что произойдет при сотнях URL? Что произойдет при сотнях производителей или потребителей? При таких масштабах подобная многозадачность становится практически обязательной. Вы не найдете ни одной действительно большой программы, которая не использует потоки для управления хотя бы частью фоновой работы. В самом деле, вы уже видели, что графическому пакету Java Swing требуется отдельный поток, обеспечивающий нормальную скорость реакции пользовательского интерфейса даже в самых малых приложениях.

Обращение к переменным классов и экземпляров из нескольких потоков

В примере с классом `SpreadSheet` мы защитили доступ к группе переменных экземпляра синхронизируемым методом, чтобы избежать изменения одной

переменной во время чтения других переменных. Мы хотели обеспечить их координацию. Но как насчет отдельных переменных? Нужно ли их синхронизировать? Как правило, нет.

Почти все операции с примитивами и ссылочными типами в Java выполняются как *атомарные*, иначе говоря, виртуальная машина выполняет их за один шаг, не оставляя возможности для коллизий двух потоков. Тем самым предотвращается обращение потока по ссылкам в тот момент, когда к ним обращаются другие потоки.

Но обратите внимание: мы сказали «почти все». Если вы внимательно прочитаете спецификацию виртуальной машины Java, то заметите, что для примитивных типов `double` и `long` атомарность операций не гарантирована. Оба этих типа представляют 64-разрядные значения.

Проблема связана с их размещением в стеке виртуальной машины Java. Возможно, в будущем спецификация будет доработана. Но пока, строго говоря, вы должны или синхронизировать обращения к переменным экземпляра `double` и `long` на уровне методов доступа, или использовать ключевое слово `volatile`, или использовать класс атомарной обертки (об этом будет рассказано далее).

Другая проблема, не связанная с атомарностью значений, — кэширование значений разными потоками виртуальной машины. Иначе говоря, даже если один поток изменил значение, может оказаться, что виртуальная машина Java в действительности не осуществит это изменение, пока не достигнет определенного состояния, называемого «барьером памяти». Чтобы предварительно решить эту проблему, можно объявить переменную с ключевым словом `volatile`. Это ключевое слово сообщает виртуальной машине, что значение может быть изменено внешними потоками, и автоматически обеспечивает синхронизацию обращений к нему. Мы говорим о «предварительном решении», потому что в многоядерных компьютерах возникают новые риски нестабильной, некорректной работы приложений. В завершающих абзацах раздела «Вспомогательные средства параллелизма», с. 330, приведены рекомендации по выбору литературы, которая пригодится, если вы намерены заняться коммерческой разработкой многопоточного кода.

Наконец, пакет `java.util.concurrent.atomic` предоставляет синхронизируемые классы-обертки для всех примитивных типов и ссылок. Обертки предоставляют не только простые операции `set()` и `get()` со значениями, но и специализированные «комбинированные» операции (такие, как `compareAndSet()`), которые выполняются как атомарные и могут использоваться для создания высокоуровневых синхронизируемых компонентов приложений. Во многих случаях классы этого пакета спроектированы специально для отображения на функциональность аппаратного уровня, и они могут быть очень эффективными.

Планирование и приоритеты

Java предоставляет ряд гарантий относительно планирования потоков. Почти всем планированием потоков занимается Java и лишь в определенной степени — приложение. Если бы создатели Java определили алгоритм планирования, это выглядело бы разумно (и упростило бы жизнь многих программистов), но единый алгоритм не обязательно подойдет для всех ролей, в которых может выступить Java. Вместо этого разработчики переложили бремя на вас: вы пишете стабильный код, работающий независимо от алгоритма планирования, и предоставляете исполнительной среде возможность выбрать оптимальный алгоритм.

Правила приоритетов, которые будут описаны ниже, тщательно сформулированы в спецификации языка Java как общие рекомендации для планирования потоков. Вы можете положиться на это поведение в целом (на статистическом уровне), но старайтесь не писать такой код, правильная работа которого зависит от конкретных особенностей планировщика. Вместо этого для координации потоков следует использовать средства контроля и синхронизации, описанные в этой главе¹.

С каждым потоком связывается значение *приоритета*. В общем случае каждый раз, когда поток с более высоким приоритетом, чем у текущего потока, становится пригодным к выполнению (запускается, перестает бездействовать или получает уведомление), он вытесняет поток с более низким приоритетом и начинает выполнение. По умолчанию потоки с одинаковым приоритетом планируются по циклической схеме. Это означает, что когда поток начинает выполняться, он продолжает работать, пока не произойдет одно из следующих событий:

- Поток переходит в состояние бездействия вызовом `Thread.sleep()` или `wait()`.
- Поток ожидает захвата блокировки для выполнения синхронизированного метода.
- Поток блокируется в ожидании ввода-вывода (например, при вызове `read()` или `accept()`).
- Поток явно уступает управление вызовом `yield()`.
- Поток прекращает работу из-за завершения его целевого метода².

¹ В книге «Java threads» Скотта Оукса (Scott Oaks) и Генри Вонга (Henry Wong), вышедшей в издательстве O'Reilly, подробно рассматриваются темы синхронизации и планирования, а также другие аспекты, связанные с потоками.

² Строго говоря, поток еще может завершиться из-за вызова устаревшего метода `stop()`, но, как мы сказали в начале главы, это не рекомендуется делать по многим причинам.

Ситуация выглядит так, как показано на рис. 9.4.

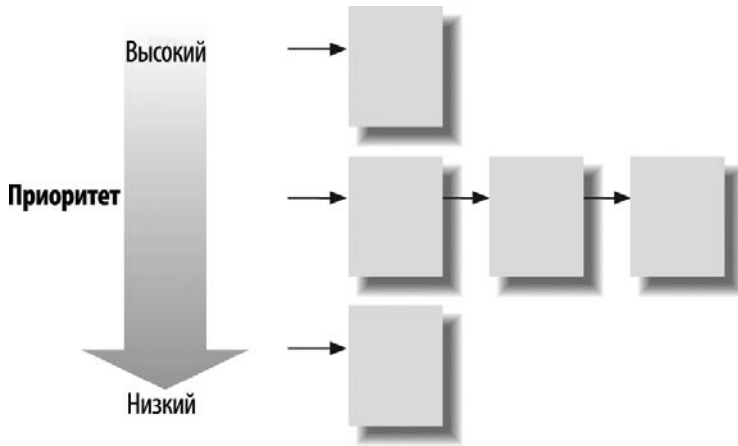


Рис. 9.4. Планирование потоков с вытеснением по циклической схеме

Состояние потока

В любой момент времени поток находится в одном из пяти обобщенных состояний, охватывающих весь его жизненный цикл и все его действия. Эти состояния определяются в перечислении `Thread.State`, а для получения информации о них используется метод `getState()` класса `Thread`:

NEW

Поток создан, но еще не запущен.

RUNNABLE

Нормальное активное состояние выполняемого потока, включающее время блокирования потока в операциях ввода-вывода (например, чтения, записи или передачи данных по сети).

BLOCKED

Поток заблокирован в ожидании входа в синхронизируемый метод или блок. Состояние включает время пробуждения потока вызовом `notify()` и пребывания в попытках повторного захвата блокировки после `wait()`.

WAITING, TIMED_WAITING

Поток ожидает другой поток вследствие вызова `wait()` или `join()`. В случае `TIMED_WAITING` ожидание ограничивается тайм-аутом.

TERMINATED

Поток завершился из-за возврата управления, исключения или остановки.

Состояние всех потоков Java (в текущей группе потоков) демонстрирует следующий фрагмент кода:

```
Thread [] threads = new Thread [64]; // Максимальное количество потоков
int num = Thread.enumerate( threads );
for( int i = 0; i < num; i++ )
    System.out.println( threads[i] + ":" + threads[i].getState() );
```

Вероятно, этот API не будет использоваться вами в повседневном программировании, но он интересен и полезен для экспериментов и для детального изучения потоков Java.

Квантование

Помимо системы приоритетов, все современные системы (кроме некоторых встроенных реализаций и «микросред» Java) реализуют *квантование* (time-slicing) потоков по времени. В системе с квантованием времени выполнение потоков усекается так, что каждый поток выполняется в течение короткого периода времени, после чего контекст переключается на следующий поток (рис. 9.5).

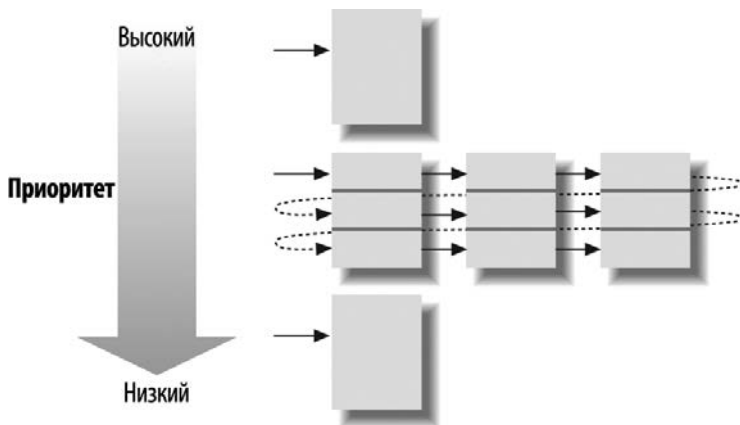


Рис. 9.5. Планирование потоков с вытеснением и квантованием по времени

В этой схеме потоки с высоким приоритетом по-прежнему вытесняют потоки с низким приоритетом. Добавление квантования распределяет вычислительные мощности между потоками с одинаковым приоритетом; на многопроцессорном компьютере потоки даже могут выполняться одновременно. Это может породить

различия в поведении тех приложений, которые неправильно используют потоки или синхронизацию.

Строго говоря, поскольку Java не гарантирует квантование по времени, вам не следует писать код, зависящий от этого типа планирования. Любая написанная вами программа должна функционировать в циклической схеме. Если вас интересует, как поступает конкретная разновидность Java, проведите 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` запускает два объекта `ShowThread`. `ShowThread` — поток, который переходит в жесткий цикл (очень плохая разновидность цикла) и выводит свое сообщение. Так как не указаны приоритеты ни для одного потока, они оба наследуют приоритет своего создателя и поэтому обладают одинаковым приоритетом. Запустив этот код, вы увидите, как устроено планирование в вашей разновидности Java. В циклической схеме будет выводиться только «Foo»; сообщение «Bar» не появится ни разу. В варианте с квантованием времени сообщения «Foo» и «Bar» должны время от времени чередоваться.

Приоритеты

Как было сказано ранее, приоритеты потоков — это всего лишь общие рекомендации относительно того, как исполнительная среда Java должна распределять время между конкурирующими потоками. К сожалению, потоки Java крайне сложно сопоставить с платформенными (низкоуровневыми) потоками компьютера, поэтому вы не можете положиться на точный смысл приоритетов. Их следует рассматривать только как рекомендации для виртуальной машины.

Поэкспериментируем с приоритетами 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 );

        foo.start();
        bar.start();
    }
}
```

Можно ожидать, что с такими изменениями в классе **Thready** поток **Bar** полностью захватит управление. При выполнении этого кода в старой реализации Java 5.0 для Solaris именно так и происходит. Но в Windows или в некоторых других старых версиях Java поведение будет другим. Кроме того, если заменить минимальный и максимальный приоритеты другими значениями, то может оказаться, что ничего не изменилось. Все эти нюансы, связанные с приоритетами и производительностью, объясняются тем, как потоки и приоритеты Java отображаются на реальные потоки операционной системы. По этой причине приоритеты потоков должны быть зарезервированы только для системы и для разработки фреймворков.

Уступка управления

Каждый раз, когда поток «засыпает», переходит в ожидание или блокируется по вводу-выводу, он уступает свой квант времени, а система планирует для выполнения другой поток. Если вы не пишете методы, использующие жесткие циклы, то все потоки должны получить возможности для выполнения. Но поток может в любой момент вызвать метод `yield()`, сообщая тем самым, что он хочет добровольно уступить свое процессорное время. Предыдущий пример можно изменить так, чтобы каждая итерация включала вызов `yield()`:

```
...
static class ShowThread extends Thread {
    ...
    public void run() {
        while ( true ) {
            System.out.println( message );
            yield();
        }
    }
}
```

Сообщения «Foo» и «Bar» должны строго чередоваться. Если у вас есть потоки, выполняющие интенсивные вычисления (или по иной причине поглощающие значительную долю процессорного времени), то вам надо поискать подходящее место для периодической уступки управления. Или можно понизить приоритет потока с интенсивными вычислениями, чтобы более важные вычисления могли периодически занимать его место.

К сожалению, спецификация языка Java очень слаба в отношении `yield()`. Это еще один аспект, который следует рассматривать не как гарантию, а как рекомендацию для оптимизации. В худшем случае исполнительная система может просто игнорировать вызовы `yield()`.

Быстродействие потоков

Способы применения потоков в приложениях, а также связанные с потоками достоинства и недостатки сильно повлияли на архитектуру многих Java API. Некоторые проблемы подробнее рассматриваются в следующих главах. А сейчас мы кратко расскажем о некоторых аспектах быстродействия потоков, а также о том, как потоки определяли форму и функциональность ряда недавних пакетов Java.

Цена синхронизации

Захват блокировок для синхронизации потоков требует времени даже при отсутствии конкуренции. В старых реализациях Java это время могло быть весьма значительным. Но в новых виртуальных машинах оно пренебрежимо мало. Тем не менее излишняя низкоуровневая синхронизация все еще может замедлять приложения из-за блокировки потоков в тех случаях, когда лучше было бы разрешить корректный одновременный доступ. Из-за этого два важных API — API коллекций Java и API графических интерфейсов Swing — специально проектировались так, чтобы избегать излишней синхронизации, передавая ее под контроль разработчика.

API коллекций из пакета `java.util` заменяет более ранние и простые агрегатные типы Java (`Vector` и `Hashtable`) более полнофункциональными, а главное, несинхронизированными типами (`List` и `Map`). API коллекций передает коду приложения возможность синхронизировать доступ к коллекциям по мере необходимости и предоставляет специальную функциональность «быстрого отказа», которая помогает обнаруживать одновременные обращения и выдавать исключения. Также предоставляются обертки синхронизации, позволяющие предоставлять безопасный доступ в старом стиле. Пакет `java.util.concurrent` включает специальные реализации `Map` и `Queue`, поддерживающие

параллельный доступ. Эти реализации идут еще дальше: они допускают высокий уровень параллельного доступа без какой-либо синхронизации со стороны пользователя.

В API Swing выбран другой подход к обеспечению скорости и безопасности. Swing требует, чтобы все изменения его компонентов (за некоторыми исключениями) осуществлялись одним потоком: главной очередью событий. Swing решает проблемы быстродействия, а также неприятные проблемы детерминизма в упорядочении событий благодаря жесткому управлению графическим интерфейсом из одного суперпотока. Приложение обращается к потоку очереди событий косвенно, ставя команды в очередь через простой интерфейс. В главе 10 мы покажем, как это делается, а в главе 11 применим новые знания в стандартной задаче реакции на получение внешних данных из сети.

Потребление ресурсов потоками

Один из фундаментальных паттернов языка Java основан на запуске множества потоков для обработки асинхронных внешних ресурсов, например подключений к сокетам. Для максимальной эффективности веб-сервер может создать отдельный поток для каждого обслуживаемого им клиентского подключения. В этом случае операции ввода-вывода могут блокироваться и перезапускаться по мере надобности. Но каким бы эффективным ни было такое решение в отношении производительности, оно очень нерационально использует ресурсы сервера. Потоки потребляют память; у каждого потока есть собственный стек для локальных переменных, а переключение между работающими потоками (переключение контекста) создает дополнительную нагрузку на процессор. Хотя потоки относительно легковесны (теоретически на большом сервере могут работать сотни и тысячи потоков), но все-таки в какой-то момент времени ресурсы, потребляемые потоками, могут вступить в противоречие с той целью, ради которой запущено так много потоков. Часто это противоречие возникает всего с несколькими десятками потоков. А решение с созданием отдельного потока для каждого клиента не всегда масштабируется.

В таких ситуациях часто создаются «пулы потоков»: фиксированный набор потоков извлекает задачи из очереди и возвращается за новыми задачами после их завершения. Такое повторное использование потоков обеспечивает стабильную масштабируемость, но раньше его было трудно эффективно реализовать на Java-серверах, потому что потоковый ввод-вывод (для таких источников, как сокеты) не в полной мере поддерживал неблокирующие операции. Пакет NIO поддерживает асинхронные каналы: неблокирующие операции чтения и записи, а также возможность проверки готовности потоков к перемещению данных. Каналы также могут асинхронно закрываться, что позволяет потокам корректно работать с ними. Пакет NIO дает возможность

создавать серверы с намного более сложными и масштабируемыми схемами многопоточности.

Пулы потоков и службы «исполнителей» (executor) заданий реализованы как часть пакета `java.util.concurrent`. Это означает, что вам не придется писать их самостоятельно. Мы кратко рассмотрим их в следующем разделе.

Вспомогательные средства параллелизма

До сих пор в этой главе мы показывали, как создавать и синхронизировать потоки на низком уровне с помощью примитивов языка Java. Пакет `java.util.concurrent` и подпакеты, появившиеся в Java 5.0, развивают эту функциональность: дополняют ее важными средствами многопоточности и систематизируют ряд основных паттернов программирования, предоставляя их стандартные реализации. Ниже перечислены эти средства, начиная с самых актуальных:

- Реализации коллекций с поддержкой потоков.

Пакет `java.util.concurrent` дополняет API коллекций Java (которому посвящена глава 7) несколькими моделями многопоточности. Во-первых, это реализации интерфейса `Queue` с ограниченными по времени ожиданием и блокировкой; во-вторых — неблокирующие, оптимизированные для параллельного доступа реализации интерфейсов `Queue` и `Map`. Пакет также добавляет в Java реализации `List` и `Set` с «копированием при записи» («copy on write»), чрезвычайно эффективные в ситуациях с доступом «почти всегда для чтения». Термины могут показаться сложными, но на самом деле эти инструменты хорошо работают в некоторых простейших ситуациях.

- Исполнители.

Исполнители (Executor) запускают задачи (включая объекты `Runnable`) и абстрагируют от программиста концепцию создания потоков и пулов. Предполагается, что исполнители станут высокоуровневой заменой идиомы создания новых потоков для обслуживания серий заданий. Наряду с Executor представлены интерфейсы `Callable` и `Future`, расширяющие `Runnable` для поддержки управления, возврата значений и обработки исключений.

- Низкоуровневые конструкции синхронизации.

Пакет `java.util.concurrent.locks` содержит набор классов (включая классы `Lock` и `Condition`), которые воспроизводят примитивы синхронизации уровня языка Java и поднимают их на уровень конкретного API. Пакет `locks` также добавляет концепцию немонопольных блокировок чтения-записи, обеспечивающих более высокий уровень параллелизма при синхронизированном доступе к данным.

- Высокоуровневые конструкции синхронизации.

К этой категории относятся классы `CyclicBarrier`, `CountDownLatch`, `Semaphore` и `Exchanger`. Эти классы реализуют распространенные паттерны синхронизации, позаимствованные из других языков и систем, и могут послужить основой для новых высокоуровневых средств.

- Атомарные операции.

Пакет `java.util.concurrent.atomic` предоставляет обертки и вспомогательные средства для атомарных (по принципу «все или ничего») операций с примитивными типами и ссылками. В их число входят простые комбинации атомарных операций (такие, как проверка значения перед присваиванием или чтение и инкремент числа) в одной операции.

Кроме возможных оптимизаций, применяемых виртуальной машиной Java для пакета атомарных операций, все эти средства реализуются на «чистом» языке Java, поверх стандартных конструкций синхронизации Java. Это означает, что они в определенном смысле существуют просто для удобства программиста и не добавляют новой функциональности в язык. Главное, они предоставляют стандартные паттерны и идиомы многопоточного программирования, делая их более безопасными и эффективными. Хорошим примером служит интерфейс `Executor`, который позволяет управлять набором задач в заранее определенной многопоточной модели, не касаясь подробностей создания потоков. Подобные высокоуровневые API упрощают программирование и повышают эффективность оптимизации в типичных случаях.

Хотя в этой главе перечисленные пакеты не рассматриваются, вы должны знать, где продолжить исследования, если тема параллелизма интересует вас или хорошо подойдет для задач, которыми вы занимаетесь на работе. Как указано в сноске в разделе «Синхронизация очереди URL-адресов», с. 318, книгу Брайана Гетца «Java Concurrency на практике» обязательно надо изучить в ходе реальных проектов. Также мы выражаем благодарность Дугу Ли (Doug Lea) — автору книги «Concurrent Programming in Java» (издательство Addison-Wesley), который возглавлял группу, добавившую эти пакеты в Java, и приложил немало усилий для их создания.

Фреймворк Swing несколько раз мимоходом упоминался в этой книге, даже в этой главе — в контексте быстрого действия потоков. Пришло время рассмотреть его более подробно.

Десктопные приложения

Своей известностью и славой язык Java в значительной мере обязан апплетам — *интерактивным* элементам веб-страниц. В наши дни это звучит обыденно, но когда-то интерактивность казалась чудом. Более того, язык Java обеспечивал кроссплатформенность: один и тот же код мог работать в системах Windows, Unix и Mac OS! В ранние версии JDK входил рудиментарный набор графических компонентов, объединенных термином AWT (Abstract Window Toolkit). «Абстрактность» в названии AWT означает, что типовые классы (`Button`, `Window` и т. д.) имеют низкоуровневые платформенные интерфейсы. Приложения AWT пишутся в виде абстрактного, кросс-платформенного кода; ваш компьютер запускает приложение и предоставляет ему конкретные платформенные компоненты.

К сожалению, это изящное сочетание абстрактного и платформенного имело существенные ограничения. В области абстрактного приходилось иметь дело с решениями из разряда «наименьшего общего кратного», предоставлявшими доступ только к тем возможностям, которые были доступны на **всех** платформах с поддержкой Java. А в платформенных реализациях даже некоторые средства, распространенные более или менее повсеместно, вели себя по-разному при выводе на экран. Многие разработчики десктопных приложений, работавшие на Java, в те дни шутили, что девиз «Написанное один раз выполняется везде» правильнее было бы изменить: «Написанное один раз приходится отлаживать везде». Пакет Swing был призван исправить это прискорбное положение дел. Он не решил всех проблем разработки кросс-платформенных приложений, зато позволил разрабатывать на Java полноценные десктопные приложения. Вы можете встретить и многие качественные проекты с открытым кодом, и даже некоторые коммерческие приложения, написанные с помощью Swing. Более того, даже интегрированная среда IntelliJ IDEA, рассматриваемая в этой книге, является Swing-приложением! Она явно не уступает платформенным интегрированным средам — как по быстродействию, так и по удобству работы в ней¹.

¹ На тот случай, если вам интересна эта тема и вы хотите увидеть, как устроено коммерческое десктопное Java-приложение, компания JetBrains опубликовала исходный код версии Community Edition (<https://oreil.ly/YleE5>).

Просматривая документацию пакета `javax.swing`¹, вы увидите, что он содержит множество классов. И при этом вам все равно будут нужны некоторые фрагменты из «древнего мира» `java.awt`. Целые книги написаны об AWT («Java AWT Reference», Zukowski, O'Reilly), о Swing («Java Swing», 2-е издание, Лой и др., O'Reilly) и даже о таких отдельных подпакетах, как Java 2D («Java 2D Graphics», Knudsen, O'Reilly). В этой главе мы ограничимся рассмотрением самых популярных компонентов: кнопок, ползунков и т. д. Мы покажем, как разместить их в окне приложения и как взаимодействовать с ними. Вы не поверите, насколько сложное приложение можно написать на основе этих простых средств. И если после прочтения книги вы займетесь разработкой десктопных приложений, вас удивит изобилие существующего Java-контента для графических интерфейсов (GUI, или просто UI). Мы постараемся разжечь ваш интерес, признавая при этом, что есть **очень много** других аспектов GUI, которые вам потом придется изучать самостоятельно. Итак, начнем наш головокружительный обзор пакета Swing!

Кнопки, ползунки и текстовые поля

С чего начать? Похоже, мы сталкиваемся с классической проблемой «курица или яйцо». Нам надо обсудить те сущности, которые должны выводиться на экран, например объекты `JLabel`, которые демонстрировались в разделе «HelloJava», с. 66. Но также необходимо рассмотреть и то, куда эти сущности помещаются. А этот вопрос заслуживает отдельного обсуждения, потому что он далеко не тривиален. В общем, берите чашку кофе, и мы начинаем. Сначала рассмотрим некоторые популярные компоненты («сущности»), затем их контейнеры и, наконец, вопрос размещения компонентов в этих контейнерах. А когда вы научитесь размещать на экране группу симпатичных виджетов, мы поговорим о том, как взаимодействовать с ними и как работают интерфейсы в многопоточном мире.

Иерархии компонентов

Как упоминалось в предыдущих главах, классы Java проектируются и расширяются в форме иерархии. На вершине иерархии классов Swing находятся классы `JComponent` и `Container` (рис. 10.1). Мы не будем подробно рассматривать эти два класса, но вам надо запомнить их имена. В них содержатся некоторые полезные атрибуты и методы, которые вам встретятся в документации Swing. Когда у вас накопится опыт программирования, в какой-то момент вам наверняка захочет-

¹ Префикс `javax` корпорация Sun изначально ввела для пакетов, которые распространялись вместе с Java, но не считались «фундаментальными». Это решение было неоднозначным, но префикс прижился и со временем стал использоваться и с другими пакетами.

ся написать собственный компонент. `JComponent` станет отличной отправной точкой. Мы проделаем это для того, чтобы реализовать очередной этап нашей игры с бросанием яблок.

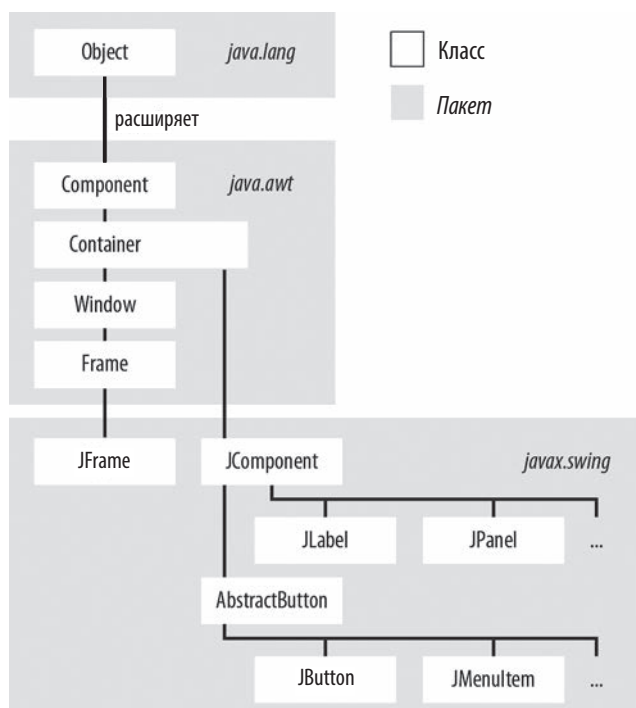


Рис. 10.1. Часть (очень небольшая) иерархии классов Swing

Мы рассмотрим многие классы, упомянутые в этой сокращенной иерархии, но вам определенно стоит обратиться к электронной документации (<https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>). Вы увидите, сколько компонентов нам пришлось пропустить при обсуждении.

Архитектура «Модель — представление — контроллер»

В основании модели Swing лежит паттерн проектирования, известный под названием «Модель — представление — контроллер» (MVC). Авторы пакета Swing основательно потрудились над последовательным применением этого паттерна, чтобы при знакомстве с новыми компонентами их поведение и использование казались вам знакомыми. Архитектура MVC стремится отделить то, что вы видите (представление), от внутреннего состояния (модели) и от со-

вокупности взаимодействий (контроллер), которые вызывают изменения в этих частях. Такое разделение обязанностей позволяет вам сосредоточиться на том, чтобы правильно разработать каждую часть. Сетевой трафик может незаметно обновлять модель. Представление может синхронизироваться через постоянные интервалы времени, благодаря чему интерфейс «не тормозит» и быстро реагирует на действия пользователя. Паттерн MVC предоставляет мощную, но хорошо управляемую архитектуру, которая подходит для разработки любых десктопных приложений.

Рассматривая нашу небольшую подборку компонентов, мы будем выделять элементы модели и представления. К подробному рассмотрению контроллеров мы перейдем в разделе «События», с. 365. Если концепция паттернов программирования вас заинтересует, самой авторитетной работой в этой области является книга Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса «Паттерны объектно-ориентированного проектирования». Детальную информацию о применении паттерна MVC в Swing вы найдете во вводной главе книги Марка Лоя (Marc Loy) и соавторов «Java Swing» (2-е издание).

Надписи и кнопки

Не приходится удивляться тому, что самый простой компонент оказывается самым популярным. Надписи (labels) используются повсеместно: для индикации функциональности, статуса вывода и т. д. Мы использовали надпись в своем первом графическом приложении в главе 2. Вы будете довольно часто использовать надписи при создании более интересных программ. Компонент `JLabel` — универсальный инструмент. Следующие примеры помогут вам понять, как использовать компонент `JLabel` и настраивать его многочисленные атрибуты. Для начала вернемся к программе `HelloJava` с несколькими модификациями:

```
package ch10;

import javax.swing.*;
import java.awt.*;

public class Labels {

    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JLabel Examples" );
        frame.setLayout(new FlowLayout());; ❶
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ❷
        frame.setSize( 300, 300 );

        JLabel basic = new JLabel("Default Label"); ❸

        frame.add(basic);

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

Самые интересные моменты этого кода:

- ❶ Выбор менеджера макета, который будет использоваться в окне.
- ❷ Выбор действия, которое будет выполняться при нажатии кнопки закрытия, предоставляемой операционной системой (в данном случае — красной точки в левом верхнем углу окна). Выбранное действие завершает работу приложения.
- ❸ Создание простой надписи.

Как видно из листинга, надпись сначала объявляется и инициализируется, а затем добавляется в окно. Будем надеяться, что эта часть выглядит знакомо. А новым будет использование экземпляра `FlowLayout`. Эта строка помогает создать снимок экрана, показанный на рис. 10.2.

Менеджеры макетов будут подробнее рассмотрены в разделе «Контейнеры и макеты», с. 353, но сейчас надо сделать первый шаг, а заодно научиться добавлять несколько компонентов в один контейнер. Класс `FlowLayout` заполняет контейнер, для чего он выравнивает компоненты по горизонтали у верхнего края, а затем добавляет их слева направо, пока в ряду не закончится свободное место, после чего переходит на следующий ряд. Такое расположение, скорее всего, не пригодится в больших приложениях, но оно идеально подойдет для быстрого размещения нескольких объектов на экране.

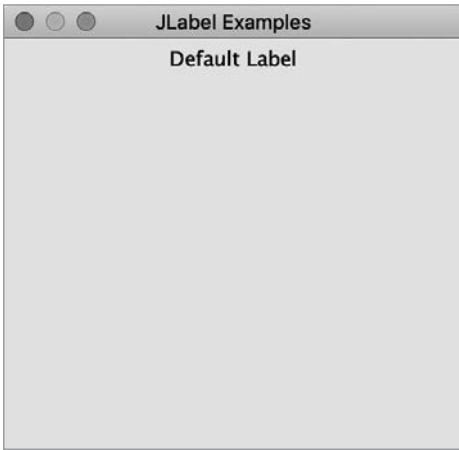
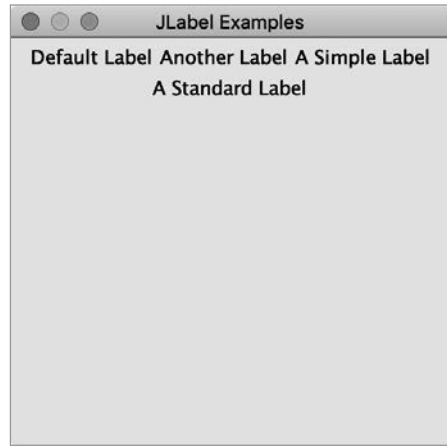
Чтобы продемонстрировать заполнение контейнера, нам нужны дополнительные надписи. Составьте еще несколько объявлений надписей и добавьте их в окно. Результат показан на рис. 10.3.

```
public class Labels {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JLabel Examples" );
        frame.setLayout(new GridLayout(0,1));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize( 300, 300 );

        JLabel basic = new JLabel("Default Label");
        JLabel another = new JLabel("Another Label");
        JLabel simple = new JLabel("A Simple Label");
        JLabel standard = new JLabel("A Standard Label");

        frame.add(basic);
        frame.add(another);
        frame.add(simple);
        frame.add(standard);

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


**Рис. 10.2.** Один простой объект JLabel**Рис. 10.3.** Несколько простых объектов JLabel

Удобно, не так ли? Еще раз подчеркнем: этот простой макет не подойдет для большинства видов контента, который встречается в коммерческих приложениях. Но он определенно полезен на первых порах. И мы должны сделать еще одно примечание, потому что эта идея встретится позднее: `FlowLayout` также обрабатывает размеры надписей. В данном примере это трудно заметить, потому что надписи по умолчанию имеют прозрачный фон. Если импортировать класс `java.awt.Color`, мы сможем воспользоваться им, чтобы сделать фон непрозрачным и назначить ему конкретный цвет:

```
// ...
JLabel basic = new JLabel("Default Label");
basic.setOpaque(true);
basic.setBackground(Color.YELLOW);
JLabel another = new JLabel("Another Label");
another.setOpaque(true);
another.setBackground(Color.GREEN);

frame.add(basic);
frame.add(another);
// ...
```

Если проделать то же самое для всех надписей, мы увидим их истинные размеры и промежутки между ними, как показано на рис. 10.4. Но если мы можем управлять цветом фона надписей, что еще можно сделать? Изменить цвет фона? (Да.) Изменить шрифт? (Да.) Изменить выравнивание? (Да.) Добавить значки? (Да.) Создать надписи с искусственным интеллектом, которые в итоге построят «Скайнет» и истребят человечество? (Возможно, но это непростая задача.) Некоторые возможные модификации изображены на рис. 10.5.

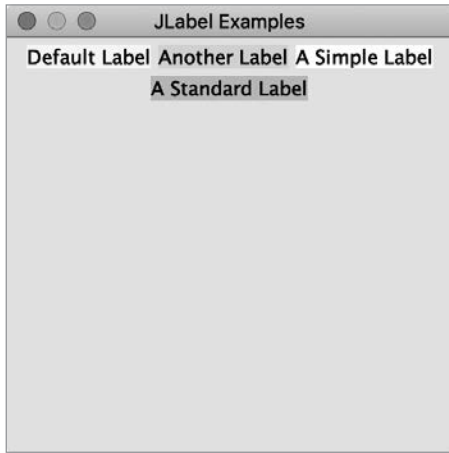


Рис. 10.4. Непрозрачные цветные надписи

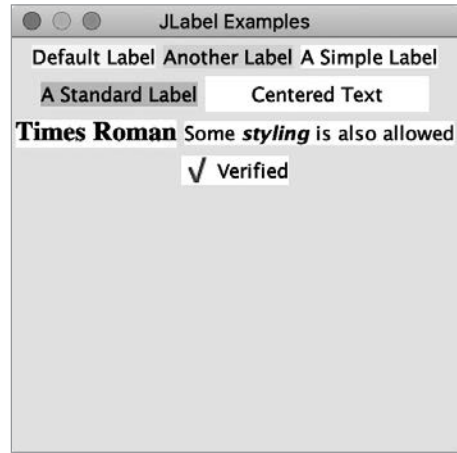


Рис. 10.5. Новые надписи с экзотическим оформлением

А теперь — исходный код, который выводит все эти разнообразные надписи:

```
// ...
JLabel centered = new JLabel("Centered Text", JLabel.CENTER);
centered.setPreferredSize(new Dimension(150, 24));
centered.setOpaque(true);
centered.setBackground(Color.WHITE);

JLabel times = new JLabel("Times Roman");
times.setOpaque(true);
times.setBackground(Color.WHITE);
times.setFont(new Font("TimesRoman", Font.BOLD, 18));

JLabel styled = new JLabel("<html>Some <b><i>styling</i></b> " +
" is also allowed</html>");
styled.setOpaque(true);
styled.setBackground(Color.WHITE);

JLabel icon = new JLabel("Verified", new ImageIcon("ch10/check.png"),
JLabel.LEFT);
icon.setOpaque(true);
icon.setBackground(Color.WHITE);

// ...
frame.add(centered);
frame.add(times);
frame.add(styled);
frame.add(icon);

// ...
```

В программе используются и другие классы: `java.awt.Font` и `javax.swing.ImageIcon`. Есть еще много возможностей, которые здесь можно было бы упомянуть, но нам нужно рассмотреть следующие компоненты Swing. Если вы хотите экспериментировать с надписями и пробовать другие варианты, описанные в документации Java, импортируйте маленькую вспомогательную программу, написанную нами для `jshell`, и поработайте с ней¹. Результаты выполнения этих нескольких строк показаны на рис. 10.6.



Рис. 10.6. Использование класса `Widget` в `jshell`

```
jshell> import javax.swing.*  
  
jshell> import java.awt.*  
  
jshell> import ch10.Widget  
  
jshell> Widget w = new Widget()  
w ==> ch10.Widget[frame0,0,23,300x300,layout=java.awt.B ... abled=true]  
  
jshell> JLabel label1 = new JLabel("Green")  
label1 ==> javax.swing.JLabel[,0,0,0x0,invalid,alignmentX=0. ... ion=CENTER]  
  
jshell> label1.setOpaque(true)  
  
jshell> label1.setBackground(Color.GREEN)
```

¹ Надо запустить `jshell` из каталога, содержащего скомпилированные файлы классов. Если вы работаете в IntelliJ IDEA, запустите терминал, выполните команду перехода в нужный каталог (`cd out/production/LearningJava5e`), затем запустите `jshell`.

```
jshell> w.add(label1)
$8 ==> javax.swing.JLabel[,0,0,0x0,...]
```

```
jshell> w.add(new JLabel("Quick test"))
$9 ==> javax.swing.JLabel[,0,0,0x0,...]
```

Итак, вы увидели, как легко создавать надписи (или другие компоненты, например кнопки, о которых речь пойдет в следующем разделе) и настраивать их параметры в интерактивном режиме. Это отличная возможность познакомиться с различными инструментами, которые будут в вашем распоряжении при создании десктопных приложений Java. Если вы часто используете класс `Widget`, то его метод `reset()` будет очень полезен. Этот метод удаляет все текущие компоненты и обновляет экран, чтобы вы могли начать все заново.

Кнопки

Еще один почти универсальный компонент, необходимый для графических приложений, — кнопка. Класс `JButton` — один из основных инструментов Swing. (В документации также описаны другие популярные виды кнопок, такие как `JCheckbox` и `JToggleButton`.) Создание кнопки мало чем отличается от создания надписи (рис. 10.7):



Рис. 10.7. Простой класс `JButton`

```
package ch10;

import javax.swing.*;
import java.awt.*;

public class Buttons {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JButton Examples" );
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
frame.setSize( 300, 200 );

JButton basic = new JButton("Try me!");
frame.add(basic);

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

Вы можете управлять цветами, выравниванием, шрифтами и прочими атрибутами кнопок точно так же, как это делается с надписями. Конечно, различие заключается в том, что вы можете нажать кнопку и отреагировать на это действие в программе, тогда как надписи в основном статичны. Попробуйте запустить этот код и нажать кнопку. Кнопка должна изменить цвет и выглядеть нажатой, хотя она еще не выполняет никаких функций в нашей программе. Хочется верить, что вы повидали немало приложений и веб-сайтов, так что кнопки и их поведение вам хорошо знакомы. Теперь мы рассмотрим еще несколько компонентов, прежде чем перейти к концепции «реагирования» на нажатие кнопки (в терминологии Swing это называется «событием»). Но если вам невтерпех, вы можете сразу перейти к разделу «События», с. 365.

Текстовые компоненты

Вслед за надписями и кнопками в рейтинге популярности идут текстовые поля. Эти элементы ввода, допускающие свободный ввод текстовой информации, встречаются практически в каждой веб-форме. В них можно вводить имена, адреса электронной почты, телефонные номера и номера кредитных карт. Все это можно делать даже на языках с составными глифами или с чтением справа налево. Невозможно представить себе современное десктопное приложение или даже веб-приложение, не поддерживающее ввод текста. Swing содержит три больших текстовых компонента: `TextField`, `TextArea` и `TextPane`; все они расширяют общего родителя `TextComponent`. `TextField` — классическое текстовое поле для ввода отдельных слов или одиночных строк. `TextArea` поддерживает возможность ввода гораздо больших фрагментов текста, разделенных на строки. `TextPane` — специализированный компонент для редактирования форматированного текста. В этой главе `TextPane` не используется, но стоит заметить, что в Swing имеются очень интересные компоненты, не требующие использования сторонних библиотек.

Текстовые поля

Рассмотрим примеры использования упомянутых компонентов в простом приложении. Мы ограничимся парой надписей и соответствующих текстовых полей `TextField` (значительно более распространенных, чем `TextArea`):

```
package ch10;

import javax.swing.*;
import java.awt.*;

public class TextInputs {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JTextField Examples" );
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize( 400, 200 );

        JLabel nameLabel = new JLabel("Name:");
        JTextField nameField = new JTextField(10);
        JLabel emailLabel = new JLabel("Email:");
        JTextField emailField = new JTextField(24);

        frame.add(nameLabel);
        frame.add(nameField);
        frame.add(emailLabel);
        frame.add(emailField);

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

Как видно из рис. 10.8, разметка текстового поля определяется количеством столбцов, заданным в конструкторе. Это не единственный способ инициализации текстового поля, но он может быть полезен при отсутствии других механизмов формирования макета, определяющих ширину поля. (Здесь `FlowLayout` нас немного подвел: надпись `Email:` оказалась не в одной строке с текстовым полем для ввода адреса. Впрочем, мы еще вернемся к макетам.) Попробуйте что-нибудь набрать на клавиатуре. Вы можете вводить и удалять текст; вырезать, копировать и вставлять его фрагменты, как и следовало ожидать; выделять мышью части текста и т. д.

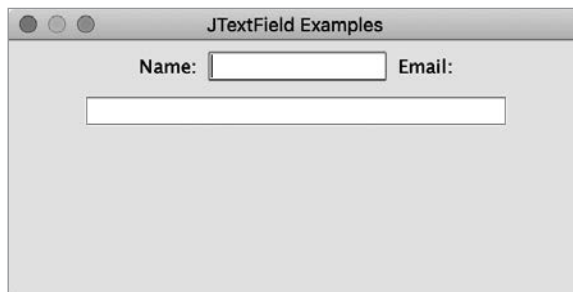


Рис. 10.8. Простые надписи и компоненты `JTextField`

Нам еще предстоит реализовать реакцию на ввод (об этом будет рассказано в разделе «События», с. 365). Впрочем, добавив текстовое поле в наше демо-приложение в `jshell`, как показано на рис. 10.9, вы сможете вызвать его метод `getText()` и узнать, какая информация содержится в текстовом поле.

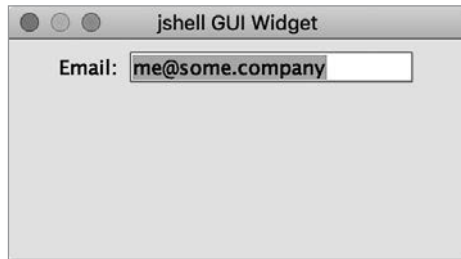


Рис. 10.9. Выделение содержимого `JTextField`

```
jshell> w.reset()

jshell> JTextField emailField = new JTextField(15)
emailField ==> javax.swing.JTextField[,0,0,0x0, ... lignment=LEADING]

jshell> w.add(new JLabel("Email:"))
$12 ==> javax.swing.JLabel[,0,0,0x0, ... sition=CENTER]

jshell> w.add(emailField)
$13 ==> javax.swing.JTextField[,0,0,0x0, ... lignment=LEADING]

// Ввод адреса электронной почты; мы ввели "me@some.company"

jshell> emailField.getText()
$14 ==> "me@some.company"
```

Свойство `text` доступно только для чтения. Чтобы изменить его содержимое на программном уровне, вызовите `setText()` для текстового поля. Например, это может быть удобно для присваивания значений по умолчанию, для автоматического форматирования таких данных, как телефонные номера, или для предварительного заполнения формы данными, собранными в сети. Опробуйте эту возможность в `jshell`.

Текстовые области

Если простых слов или даже длинных URL-адресов недостаточно, то вы, скорее всего, обратитесь к компоненту `JTextArea`, который предоставляет достаточно места для ввода. Вы можете создать пустую текстовую область с помощью конструктора, аналогичного тому, который использовался для `JTextField`, но на этот раз с передачей не только количества столбцов, но и количества строк. Код

добавления текстовой области в демоприложение приводится ниже, а результат показан на рис. 10.10:

```
JLabel bodyLabel = new JLabel("Body:");
JTextArea bodyArea = new JTextArea(10,30);

frame.add(bodyLabel);
frame.add(bodyArea);
```

Как видите, у нас хватает места для нескольких строк текста. Запустите эту новую версию и опробуйте ее. Что произойдет, если вы введете текст с выходом за конец строки? Что происходит при нажатии клавиши Enter (Return)? Будем надеяться, что поведение компонента будет знакомым. Ниже мы покажем, как скорректировать это поведение, но сейчас нужно указать, что вы по-прежнему можете получить доступ к содержимому текстовой области (как и в случае с текстовым полем).



Рис. 10.10. Добавление JTextArea

Добавим текстовую область в наше приложение в jshell:

```
jshell> w.reset()

jshell> w.add(new JLabel("Body:"))
$16 ==> javax.swing.JLabel[,0,0,0x0, ... ition=CENTER]

jshell> JTextArea bodyArea = new JTextArea(5,20)
bodyArea ==> javax.swing.JTextArea[,0,0,0x0, ... word=false,wrap=false]

jshell> w.add(bodyArea)
$18 ==> javax.swing.JTextArea[,0,0,0x0, ... lse,wrap=false]

jshell> bodyArea.getText()
$19 ==> "This is the first line.\nThis should be the second.\nAnd the third..."
```


Превосходно! Вы видите, что клавиша Enter (Return), нажатая для генерирования трех строк (рис. 10.11), кодируется символом \n в полученной строке.

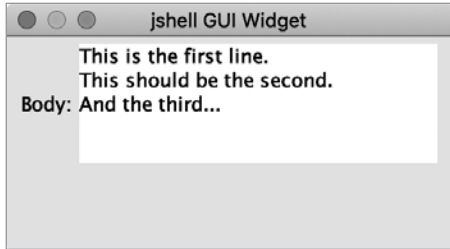


Рис. 10.11. Получение содержимого JTextArea

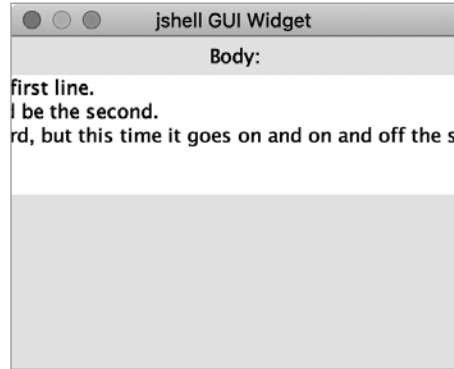


Рис. 10.12. Слишком длинная строка в простом компоненте JTextArea

А что будет, если вы захотите ввести очень длинное предложение, выходящее за ширину строки? Вероятно, получится странная текстовая область, которая расширяется до границ нашего окна и даже уходит за его пределы (рис. 10.12).

Чтобы исправить это недоразумение с размерами, обратите внимание на пару свойств JTextArea, приведенных в табл. 10.1.

Таблица 10.1. Свойства JTextArea, относящиеся к переносу

Свойство	По умолчанию	Описание
lineWrap	false	Флаг переноса строк, выходящих за ширину области
wrapStyleWord	false	Флаг разрыва строк по границам строк или символов (если используется перенос строк)

Начнем с нуля и включим перенос текста по словам. Для этого можно использовать вызов `setLineWrap(true)`, но его будет недостаточно. В дополнение следует включить вызов `setWrapStyleWord(true)`, чтобы не разрывать слова при переносах строк. Результат показан на рис. 10.13.

Вы можете опробовать программу самостоятельно в jshell или в вашем собственном приложении, если хотите убедиться в том, что третья строка действительно переносится. Обратите внимание: в тексте, полученном из объекта `bodyArea`, в третьей строке между вторым «on» и «but» не должно быть разрыва строки (\n).

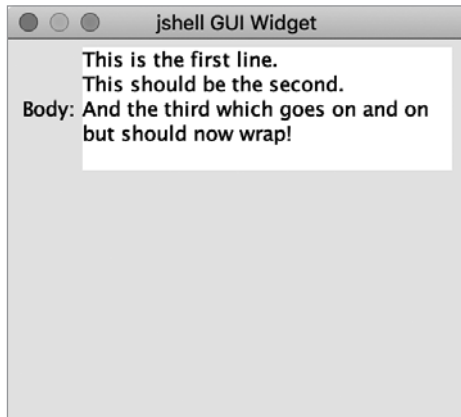


Рис. 10.13. Перенос строк в простом компоненте JTextArea

Прокрутка текста

Итак, мы решили проблему, которая возникает, если одна строка содержит слишком много символов. Но что происходит, если вся текстовая область содержит слишком много строк? Если ничего не предпринять, то `JTextArea` выполняет странный трюк «увеличиваться в размерах, пока возможно» (рис. 10.14).

Для решения этой проблемы нам понадобится помощь со стороны стандартного вспомогательного компонента Swing: `JScrollPane`. Это контейнер общего назначения, который упрощает отображение больших компонентов в ограниченном пространстве. Чтобы показать, насколько это просто, исправим нашу текстовую область¹:

```
jshell> w.remove(bodyArea); // Мы можем начать с новой текстовой области

jshell> bodyArea = new JTextArea(5,20)
bodyArea ==> javax.swing.JTextArea[,0,0,0x0,invalid,word=false,wrap=false]

jshell> w.add(new JScrollPane(bodyArea))
$17 ==> javax.swing.JScrollPane[,47,5,244x84, ... ortBorder=]
```

¹ При создании компонентов Swing для использования в примерах `jshell` мы опускаем большую часть получаемого вывода. `jshell` выводит большой объем информации о каждом компоненте, но если объем информации выходит за пределы разумного, то информация заменяется многоточием. Если во время экспериментов вы видите слишком подробную информацию об атрибутах элемента, не беспокойтесь — это нормально. Мы хотели сохранить компактность текста и решили пропустить фрагменты, не относящиеся к теме.

На рис. 10.15 видно, что компонент уже не выходит за границы окна. Также у компонента сбоку и снизу появились стандартные полосы прокрутки. Если вам не нужно ничего, кроме простой прокрутки, — все готово! Но как и у других компонентов Swing, у `JScrollPane` есть много настроек, которые вы можете регулировать по мере надобности. Большая их часть здесь не используется, но мы хотим показать, как выбрать стандартную конфигурацию для текстовых областей: перенос строк (с разбивкой по словам) с вертикальной прокруткой, но без горизонтальной прокрутки.

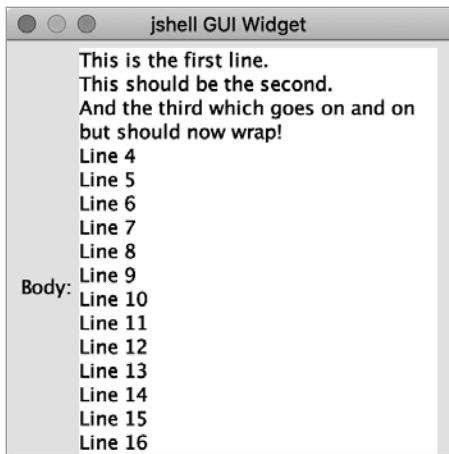


Рис. 10.14. Слишком много строк в простом компоненте `JTextArea`

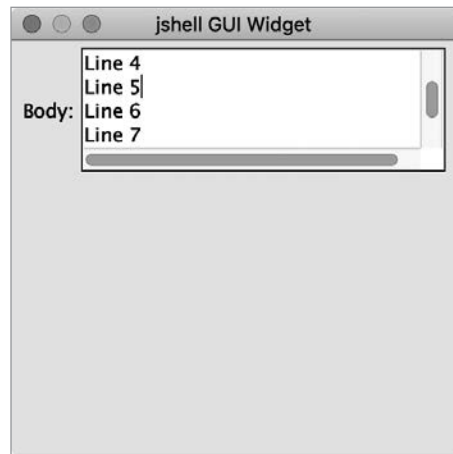


Рис. 10.15. Слишком много строк в компоненте `JTextArea`, встроенном в `JScrollPane`

Полученная текстовая область должна выглядеть так, как показано на рис. 10.16.

```
JLabel bodyLabel = new JLabel("Body:");
JTextArea bodyArea = new JTextArea(10,30);
bodyArea.setLineWrap(true);
bodyArea.setWrapStyleWord(true);
JScrollPane bodyScroller = new JScrollPane(bodyArea);
bodyScroller.setHorizontalScrollBarPolicy(
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
bodyScroller.setVerticalScrollBarPolicy(
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

frame.add(bodyLabel);
// Внимание: мы не добавляем компонент bodyArea,
// он уже находится в bodyScroller
frame.add(bodyScroller);
```

Ура! Вы получили представление о самых популярных компонентах Swing: надписях, кнопках и текстовых полях. Впрочем, наше описание этих компо-

нентов было в лучшем случае поверхностным. Все они обладают множеством разных атрибутов, которые легко настраиваются вызовами методов — примерно так, как мы вызывали метод `setBackground()` для экземпляров `JLabel` в разделе «Надписи и кнопки», с. 335. Полистайте документацию Java и поэкспериментируйте с каждым из этих компонентов в `jshell` или в вашем собственном демо-приложении. Умение работать с компонентами пользовательского интерфейса зависит от практики. Мы определенно рекомендуем обращаться к другим книгам и к онлайн-руководствам, если вы пишете десктопные приложения на работе или просто для друзей. Но ничто не заменит времени, проведенного за клавиатурой, когда вы лично работаете над приложением и исправляете неизбежные ошибки.

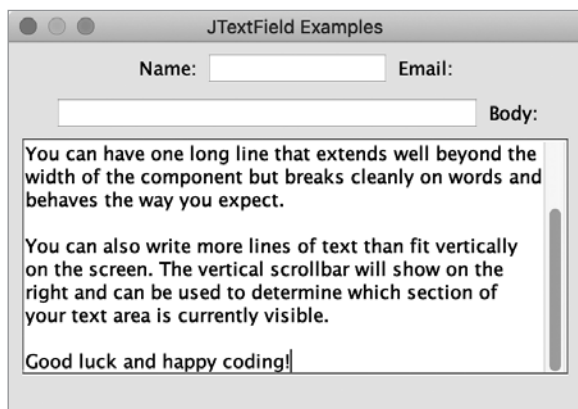


Рис. 10.16. Правильная конфигурация `JTextArea` в `JScrollPane`

Другие компоненты

Если вы уже заглядывали в документацию пакета `javax.swing`, то вам известно, что существуют десятки других компонентов, которыми вы можете пользоваться в своих приложениях. В этом большом списке есть несколько компонентов, которые заслуживают особого внимания¹.

`JSlider`

Ползунок (`slider`) — удобное и эффективное средство ввода для диапазонов значений. Вероятно, вы встречали ползунки в таких важных элементах интерфейса, как средства выбора размера шрифта или цвета (диапазоны красного, зеленого

¹ В интернете есть много проектов с открытым кодом для поддержки таких возможностей, как цветовое выделение синтаксиса, различные средства выделения текста и специализированные средства ввода (например, компоненты для выбора даты или времени).

и синего), выбора масштаба и т. д. Ползунки идеально подойдут для выбора угла и силы броска в нашей игре с бросанием яблок. Углы лежат в диапазоне от 0 до 180, а значения силы — от 0 до 20 (максимум выбран произвольно). На рис. 10.17 показаны ползунки в действии; пока несущественно, как мы добавили их в игру.

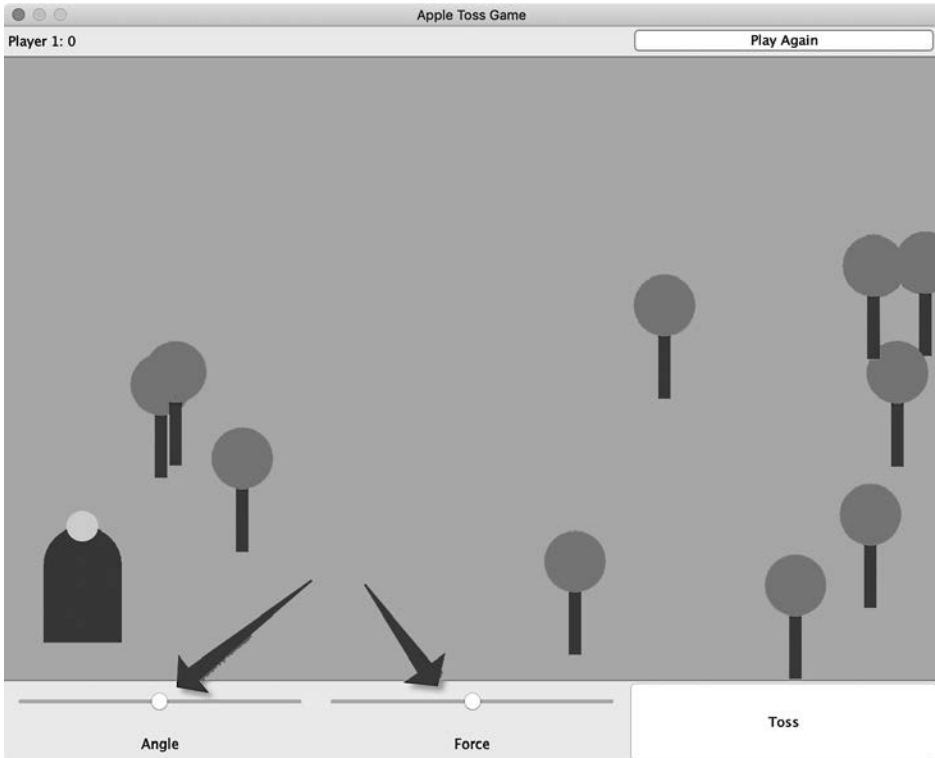


Рис. 10.17. Использование JSlider в игре с бросанием яблок

Для создания нового ползунка обычно предоставляются три значения: минимум (для угла броска мы выбрали 0), максимум (мы выбрали 180) и исходное значение (мы выбрали середину диапазона, то есть 90). Добавление этого ползунка в «песочнице» jshell выглядит так:

```
jshell> w.reset()

jshell> JSlider slider = new JSlider(0, 180, 90);
slider ==> javax.swing.JSlider[,0,0,0x0, ... ks=false,snapToValue=true]

jshell> w.add(slider)
$20 ==> javax.swing.JSlider[,0,0,0x0, ... alue=true]
```

Переместите ползунок, как показано на рис. 10.18, и проверьте его текущее значение методом `getValue()`:

```
jshell> slider.getValue()
$21 ==> 112
```



Рис. 10.18. Простой компонент `JSlider` в `jshell`

В разделе «События», с. 365, мы покажем, как получать эти значения, когда пользователь изменяет их в реальном времени.

Из документации по конструкторам `JSlider` можно узнать, что для представления минимального и максимального значений используются целые числа. Кроме того, метод `getValue()` тоже возвращает целое число. Если вам нужны дробные значения, то вам придется реализовать соответствующие вычисления самостоятельно. Например, ползунок силы броска в нашей игре мог бы поддерживать более 21 дискретного уровня. В таких ситуациях обычно создается ползунок со значительно большим диапазоном, после чего текущее значение делится на соответствующий масштабный коэффициент.

```
jshell> JSlider force = new JSlider(0, 200, 100)
force ==> javax.swing.JSlider[,0,0,0x0, ... ks=false,snapToValue=true]
```

```
jshell> w.add(force)
$23 ==> javax.swing.JSlider[,0,0,0x0,invalid ... alue=true]
```

```
jshell> force.getValue()
$24 ==> 68
```

```
jshell> float myForce = force.getValue() / 10.0f;
myForce ==> 6.8
```

JList

Если вы работаете с диапазоном значений, которые не являются обычными последовательными целыми числами, то вам отлично подойдет «списковый» GUI-элемент. `JList` реализует этот тип элемента ввода в Swing. В нем можно разрешить одиночное или множественное выделение, а если вы поглубже разберетесь в Swing, то сможете программировать даже нестандартные представления для вывода элементов списка с дополнительной информацией. (Например, списки значков, или значков и текста, или многострочного текста и т. д.)

В отличие от других компонентов, встречавшихся нам ранее, `JList` требует большего объема информации при инициализации. Для создания полезного компонента списка вам понадобится один из конструкторов, получающих данные, которые должны отображаться в списке. Простейший конструктор такого рода получает массив `Object`. Хотя можно передать массив произвольных объектов, по умолчанию `JList` будет выводить в списке результаты вызова методов `toString()` переданных объектов. Создание списков на базе массивов объектов `String` встречается очень часто и приводит к ожидаемым результатам. На рис. 10.19 показан простой список городов.



Рис. 10.19. Простой компонент `JList` с четырьмя городами в `jshell`

```
jshell> w.reset()

jshell> String[] cities = new String[] { "Atlanta", "Boston", "Chicago",
    "Denver" };
cities ==> String[4] { "Atlanta", "Boston", "Chicago", "Denver" }

jshell> JList cityList = new JList<String>(cities);
cityList ==> javax.swing.JList[,0,0,0x0, ... ,layoutOrientation=0]
```

```
jshell> w.add(cityList)
$29 ==> javax.swing.JList[,0,0,0x0,invalid ... ation=0]
```

Обратите внимание: в конструкторе используется информация о типе `<String>`, как при создании объектов коллекций вроде `ArrayList` (см. «Ограничения типов», с. 242). Так как пакет `Swing` появился задолго до обобщений, в книгах и в коде из интернета могут встретиться примеры, в которых информация о типе не добавляется. Как и в случае с коллекциями, это не мешает компиляции или запуску вашего кода, но вы увидите те же предупреждения о непроверяемых преобразованиях во время компиляции.

По аналогии с текущим значением ползунка вы можете получить выбранный элемент (или элементы) списка. Для этого используются следующие четыре метода:

- `getSelectedIndex()` — для списков с одиночным выбором, возвращает `int`.
- `getSelectedIndices()` — для списков с множественным выбором, возвращает массив `int`.
- `getSelectedValue()` — для списков с одиночным выбором, возвращает объект.
- `getSelectedValues()` — для списков с множественным выбором, возвращает массив объектов.

Разумеется, выбор зависит от того, что для вас важнее — порядковые номера выбираемых элементов или их фактические значения. Продолжая эксперименты со списком городов в `jshell`, мы можем извлечь выделенный в списке город следующим образом:

```
jshell> cityList.getSelectedIndex()
$31 ==> 2

jshell> cityList.getSelectedIndices()
$32 ==> int[1] { 2 }

jshell> cityList.getSelectedValue()
$33 ==> "Chicago"

jshell> cities[cityList.getSelectedIndex()]
$34 ==> "Chicago"
```

Вероятно, для больших списков понадобится полоса прокрутки. `Swing` стремится к повторному использованию фрагментов в своем коде, поэтому неудивительно, что с `JList` можно использовать `JScrollPane`, как это делалось с текстовыми областями в разделе «Прокрутка текста», с. 346.

Контейнеры и макеты

Вы освоили уже целый список компонентов. А ведь это всего лишь малое подмножество виджетов, доступных для ваших графических приложений! Но остальные компоненты Swing остаются вам для самостоятельного изучения — когда вы освоитесь с Java в целом и начнете проектировать конкретные программные решения для реальных задач. В этом разделе мы сосредоточимся на размещении групп компонентов в полезных комбинациях. Эти компоненты размещаются в *контейнерах*, поэтому начнем с рассмотрения самых популярных контейнеров.

JFrame и JWindow

Каждому десктопному приложению необходимо как минимум одно *окно*. Этот термин появился задолго до Swing, и на окнах основаны почти все графические интерфейсы в больших операционных системах. Swing предоставляет низкоуровневый класс `JWindow`, который может вам понадобиться для «универсальных» окон. Но скорее всего, вы будете создавать свои приложения во *фреймах* — так иногда называют «готовые» окна, формируемые классом `JFrame`. Например, наше первое приложение в главе 2 было сделано на основе `JFrame`. На рис. 10.20 показано место `JFrame` в иерархии классов. Мы будем придерживаться базовых возможностей `JFrame`, но по мере расширения функциональности ваших приложений вы, вероятно, захотите создавать собственные окна с элементами из более высоких уровней иерархии.

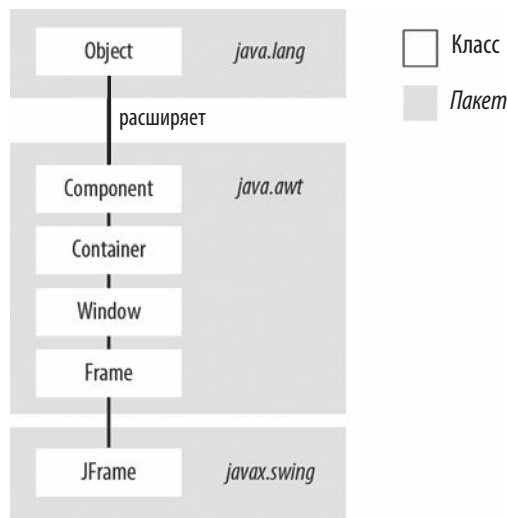


Рис. 10.20. Иерархия классов `JFrame`

Вернемся к нашему первому графическому приложению и повнимательнее рассмотрим к тому, что же происходит в создаваемом объекте `JFrame`:

```
import javax.swing.*;

public class HelloJavaAgain {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "Hello, Java!" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize( 300, 300 );

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

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

Строка, передаваемая конструктору `JFrame`, становится заголовком окна. Затем мы задаем несколько конкретных свойств объекта. При закрытии окна пользователем программа должна завершиться. (Кому-то это покажется очевидным, но в сложных приложениях бывает много окон: например, палитры с инструментами или окна разных документов. Закрытие окна в таких приложениях может и не означать завершения.) Затем мы выбираем начальный размер окна и добавляем компонент надписи; в свою очередь, `JFrame` помещает надпись на свою *панель содержимого* (*content pane*). После добавления компонента мы делаем окно видимым. Результат показан на рис. 10.21.

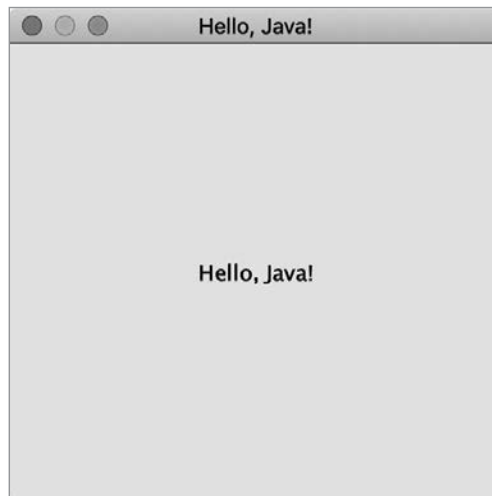


Рис. 10.21. Простой компонент `JFrame` с надписью

Этот базовый процесс лежит в основе каждого приложения Swing. Функциональность вашего приложения связана с тем, что вы будете делать с панелью содержимого. Но что представляет собой панель содержимого? Оказывается, `JFrame` использует собственный компонент-контейнер — экземпляр `JPanel` (о классе `JPanel` мы расскажем в следующем разделе). Внимательно просмотрев документацию `JFrame`, вы увидите, что панелью содержимого можно назначить любой объект, производный от `java.awt.Container`, но пока мы ограничимся панелью по умолчанию. Также можно заметить, что для добавления надписи используется сокращенный синтаксис. Версия `add()` класса `JFrame` вызывает метод `add()` панели содержимого. Например, можно было использовать следующий фрагмент:

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

Тем не менее в классе `JFrame` не предусмотрены сокращения для всех возможных операций с панелью содержимого. Прочитайте документацию и используйте сокращенную запись, если она существует. Если ее нет, вы всегда можете получить ссылку вызовом `getContentPane()`, а затем настроить или оптимизировать полученный объект.

JPanel

Класс `JPanel` — самый популярный контейнер в Swing. Он является компонентом, как и `JButton` или `JLabel`, поэтому панели могут содержать другие панели. Такое вложение часто играет важную роль в визуальной структуре приложения. Например, можно создать на панели инструментов отдельную панель `JPanel` для кнопок форматирования, которую пользователь сможет передвигать так, как ему удобно.

`JPanel` дает вам возможность добавлять и удалять компоненты с экрана. (Точнее, методы добавления и удаления наследуются от класса `Container`, но мы обращаемся к ним через объекты `JPanel`.) Также можно перерисовать панель вызовом `repaint()`, если что-то изменилось и вы хотите перерисовать пользовательский интерфейс. Эффект вызова методов `add()` и `remove()` в `jshell` показан на рис. 10.22:

```
jshell> Widget w = new Widget()
w ==> ch10.Widget[frame0,0,23,300x300, ... tPaneCheckingEnabled=true]

jshell> JLabel emailLabel = new JLabel("Email:")
emailLabel ==> javax.swing.JLabel[,0,0,0x0, ... extPosition=CENTER]

jshell> JTextField emailField = new JTextField(12)
emailField ==> javax.swing.JTextField[,0,0,0x0, ... talAlignment=LEADING]
```

```

jshell> JButton submitButton = new JButton("Submit")
submitButton ==> javax.swing.JButton[,0,0,0x0, ... aultCapable=true]

jshell> w.add(emailLabel);
$8 ==> javax.swing.JLabel[,0,0,0x0, ... ition=CENTER]
// Левый скриншот на рис. 10.22

jshell> w.add(emailField)
$9 ==> javax.swing.JTextField[,0,0,0x0, ... nment=LEADING]

jshell> w.add(submitButton)
$10 ==> javax.swing.JButton[,0,0,0x0, ... pable=true]
// Средний скриншот

jshell> w.remove(emailLabel)
// Правый скриншот

```

Попробуйте сами! Впрочем, в большинстве приложений компоненты не добавляются и не удаляются произвольно. Обычно пользователь формирует интерфейс, добавляя то, что ему необходимо, а потом оставляет его без изменений. Возможно, в ходе настройки вы заблокируете или разблокируете некоторые кнопки, но не стоит удивлять пользователя исчезновением частей интерфейса или внезапным появлением новых элементов.



Рис. 10.22. Добавление и удаление компонентов в JPanel

Менеджеры макетов

К важнейшим свойствам JPanel в Swing (или любого другого потомка Container) относится взаимное расположение добавленных в контейнер элементов в сочетании с их размерами. Применительно к пользовательским интерфейсам это называется *макетом* (*layout*) контейнера. В Java есть несколько *менеджеров макетов* (*layout managers*), которые помогут вам достигать желаемых результатов.

BorderLayout

Менеджер `FlowLayout` вы уже видели в действии (по крайней мере, в горизонтальной ориентации; один из его конструкторов может создавать столбец компонентов). Также вы использовали другой менеджер макета, хотя могли и не знать об этом. Дело в том, что панель содержимого `JFrame` по умолчанию использует режим `BorderLayout`. На рис. 10.23 изображены пять областей, находящихся под управлением `BorderLayout`, с указанием их имен. Обратите внимание: ширина областей `NORTH` и `SOUTH` определяется окном приложения, но им назначается минимальная высота, необходимая для размещения надписи. Аналогичным образом области `EAST` и `WEST` заполняют вертикальный промежуток между областями `NORTH` и `SOUTH`, но им назначается минимальная необходимая ширина, а все оставшееся пространство заполняется по горизонтали и вертикали областью `CENTER`.

```
import java.awt.*;
import javax.swing.*;

public class BorderLayoutDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame("BorderLayout Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);

        JLabel northLabel = new JLabel("Top - North", JLabel.CENTER);
        JLabel southLabel = new JLabel("Bottom - South", JLabel.CENTER);
        JLabel eastLabel = new JLabel("Right - East", JLabel.CENTER);
        JLabel westLabel = new JLabel("Left - West", JLabel.CENTER);
        JLabel centerLabel = new JLabel("Center (everything else)",
JLabel.CENTER);

        // Изменить цвета надписей, чтобы их границы были лучше видны
        northLabel.setOpaque(true);
        northLabel.setBackground(Color.GREEN);
        southLabel.setOpaque(true);
        southLabel.setBackground(Color.GREEN);
        eastLabel.setOpaque(true);
        eastLabel.setBackground(Color.RED);
        westLabel.setOpaque(true);
        westLabel.setBackground(Color.RED);
        centerLabel.setOpaque(true);
        centerLabel.setBackground(Color.YELLOW);

        frame.add(northLabel, BorderLayout.NORTH);
        frame.add(southLabel, BorderLayout.SOUTH);
        frame.add(eastLabel, BorderLayout.EAST);
        frame.add(westLabel, BorderLayout.WEST);
        frame.add(centerLabel, BorderLayout.CENTER);

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



Рис. 10.23. Области, доступные для BorderLayout

Обратите внимание: метод `add()` в этом случае получает дополнительный аргумент, который передается менеджеру макета. Не всем менеджерам нужен этот аргумент, как было показано на примере `FlowLayout`.

В следующем примере вложенные объекты `JPanel` будут очень удобны: главное приложение в `JPanel` — в центре, панель инструментов в `JPanel` — у верхнего края, строка состояния в `JPanel` — у нижнего края, менеджер проектов в `JPanel` — слева и т. д. `BorderLayout` определяет эти области по именам, соответствующим сторонам света на компасе. На рис. 10.24 приведен очень простой пример вложения контейнеров. В центре создается текстовая область для ввода длинного сообщения, а на панели в нижней части размещаются кнопки. Напомним, что без событий, которые будут рассматриваться в следующем разделе, ни одна кнопка ничего не будет делать, но сейчас мы хотим только показать работу с несколькими контейнерами. И при желании вы можете продолжать вкладывать объекты `JPanel`; только проследите за тем, чтобы ваша иерархия была наглядной и понятной. Иногда более удачный выбор макета верхнего уровня упрощает сопровождение вашего приложения и повышает его быстродействие.

```
public class NestedPanelDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame("Nested Panel Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);

        // Создать текстовую область и добавить ее в центр
        JTextArea messageArea = new JTextArea();
        frame.add(messageArea, BorderLayout.CENTER);

        // Создать контейнер для кнопок
        JPanel buttonPanel = new JPanel(new FlowLayout());

        // Создать кнопки
        JButton sendButton = new JButton("Send");
        JButton saveButton = new JButton("Save");
```

```
    JButton resetButton = new JButton("Reset");
    JButton cancelButton = new JButton("Cancel");

    // Добавить кнопки в контейнер
    buttonPanel.add(sendButton);
    buttonPanel.add(saveButton);
    buttonPanel.add(resetButton);
    buttonPanel.add(cancelButton);

    // И наконец, добавить контейнер для кнопок в нижнюю
    // область окна
    frame.add(buttonPanel, BorderLayout.SOUTH);

    frame.setVisible(true);
}
```



Рис. 10.24. Простой пример вложения контейнеров

В этом примере надо обратить внимание на два момента. Во-первых, мы не указали количество строк или столбцов при создании объекта `JTextArea`. В отличие от `FlowLayout`, `BorderLayout` задает размер своих компонентов, когда это возможно. Для верхней и нижней области это означает использование собственной высоты компонента (по аналогии с `FlowLayout`), с последующим выбором ширины компонента для заполнения окна. Боковые области используют ширину своих компонентов, но затем назначают высоту. Компонент в центре (текстовая область в нашем примере) получает свою ширину и высоту от `BorderLayout`.

Второй момент может показаться очевидным, но мы все равно хотим привлечь к нему ваше внимание. При добавлении в окно объектов `messageArea` и `buttonPanel` мы задаем дополнительный аргумент метода `add()` из `JFrame`, указывающий на местоположение. Но при добавлении кнопок в `buttonPanel` используется более простая версия `add()` только с одним аргументом. Разные вызовы `add()` привязываются к контейнеру, который выполняет вызовы, а затем передают аргументы, соответствующие менеджеру макета этого контей-

нера. Таким образом, хотя `buttonPanel` находится в области `SOUTH`, компонент `saveButton` и его «родственники» не знают об этой подробности, да им и не нужно это знать.

GridLayout

Часто требуется, чтобы компоненты или надписи располагались симметрично. Представьте привычные кнопки «Да», «Нет» и «Отмена», сдвинутые в нижнюю часть диалогового окна. (Swing позволяет создавать такие окна, но подробнее об этом — в разделе «Модальные и всплывающие окна», с. 373.) Класс `GridLayout` — один из ранних менеджеров макетов — упрощает равномерное симметричное размещение. Для начала применим `GridLayout` к кнопкам из предыдущего примера. Потребуется изменить всего одну строку:

```
// Создать контейнер для кнопок
// Старая версия: JPanel buttonPanel = new JPanel(new FlowLayout());
JPanel buttonPanel = new JPanel(new GridLayout(1,0));
```

Вызовы `add()` остались неизменными; никакой отдельный аргумент для ограничения не нужен. Результат показан на рис. 10.25.

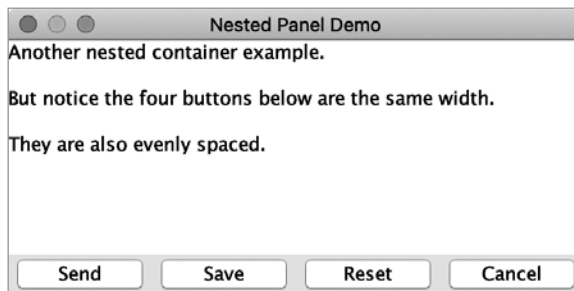


Рис. 10.25. Использование `GridLayout` для расположения кнопок в ряду

Как видно из рис. 10.25, все кнопки в `GridLayout` одинаковы по размеру, независимо от того, что на них написано (надпись `Cancel` самая длинная). При создании менеджера макета мы сообщили, что нам нужен ровно один ряд кнопок при отсутствии ограничений (0) на количество столбцов. Впрочем, как следует из названия, таблица компонентов может быть двумерной, а вы можете точно задать нужное количество строк и столбцов. На рис. 10.26 изображена клавиатура классического кнопочного телефона.

```
public class PhoneGridDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame("Nested Panel Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



```
frame.setSize(200, 300);  
  
// Создать контейнер для клавиатуры телефона  
JPanel phonePad = new JPanel(new GridLayout(4,3));  
  
// Создать и добавить 12 кнопок, слева направо, сверху вниз  
phonePad.add(new JButton("1"));  
phonePad.add(new JButton("2"));  
phonePad.add(new JButton("3"));  
  
phonePad.add(new JButton("4"));  
phonePad.add(new JButton("5"));  
phonePad.add(new JButton("6"));  
  
phonePad.add(new JButton("7"));  
phonePad.add(new JButton("8"));  
phonePad.add(new JButton("9"));  
  
phonePad.add(new JButton("*"));  
phonePad.add(new JButton("0"));  
phonePad.add(new JButton("#"));  
  
// Клавиатура добавляется в центр окна приложения  
frame.add(phonePad, BorderLayout.CENTER);  
frame.setVisible(true);  
}  
}
```

При добавлении кнопок слева направо и сверху вниз результат будет выглядеть так, как показано на рис. 10.26.



Рис. 10.26. Клавиатура телефона в виде двумерного табличного макета

Это очень удобно и просто, если вам нужны идеально симметричные элементы. А если они должны быть **не совсем** симметричными? Представьте веб-форму со столбцом надписей слева и столбцом текстовых полей справа. Бесспорно, `GridLayout` справится с такой формой, но обычно надписи короткие и простые, а текстовые поля имеют большую ширину и вмещают больше текстового ввода от пользователя. Как сделать такой макет?

GridBagLayout

Если вы хотите создать более интересный макет, но при этом избежать возни с множеством вложенных панелей, одним из возможных вариантов является `GridBagLayout`. Он настраивается чуть сложнее, но позволяет использовать довольно сложные макеты с эстетичным выравниванием и настройкой размеров элементов. По аналогии с `BorderLayout` компоненты добавляются с дополнительным аргументом. Однако в аргументе `GridBagLayout` передается полнофункциональный объект `GridBagConstraints` вместо простого объекта `String`.

«Сетка», или «таблица», в `GridBagLayout` представляет собой прямоугольный контейнер, разделенный на ячейки по строкам и столбцам. Каждая строка может иметь собственную высоту, каждый столбец — собственную ширину, а компоненты могут занимать любую прямоугольную группу ячеек. Мы можем воспользоваться этой гибкостью для создания игрового интерфейса с одним компонентом `JPanel` вместо нескольких вложенных панелей. На рис. 10.27 показан один из способов разделения экрана на четыре строки и три столбца (с последующим размещением компонентов).

На иллюстрации видно, что строки отличаются по высоте, а столбцы — по ширине. Заметьте, что некоторые компоненты занимают более одной ячейки. Такой вариант размещения подходит не для каждого приложения, но он эффективен и работает во многих пользовательских интерфейсах, которым недостаточно простого макета.

Чтобы написать приложение на базе `GridBagLayout`, вам понадобится хранить пару ссылок при добавлении компонентов. Начнем с подготовки таблицы:

```
public static final int SCORE_HEIGHT = 30;
public static final int CONTROL_WIDTH = 300;
public static final int CONTROL_HEIGHT = 40;
public static final int FIELD_WIDTH = 3 * CONTROL_WIDTH;
public static final int FIELD_HEIGHT = 2 * CONTROL_HEIGHT;
public static final float FORCE_SCALE = 0.7f;

GridBagLayout gameLayout = new GridBagLayout();

gameLayout.columnWidths = new int[]
    { CONTROL_WIDTH, CONTROL_WIDTH, CONTROL_WIDTH };
gameLayout.rowHeights = new int[]
    { SCORE_HEIGHT, FIELD_HEIGHT, CONTROL_HEIGHT, CONTROL_HEIGHT };

JPanel gamePane = new JPanel(gameLayout);
```

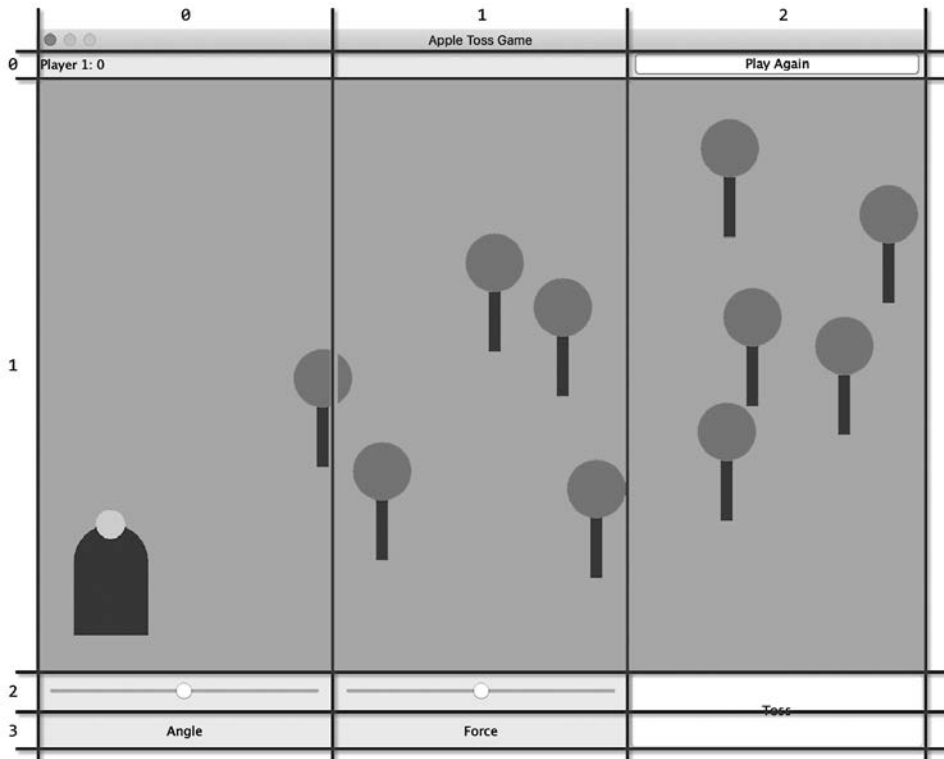


Рис. 10.27. Пример сетки для GridBagLayout

Превосходно! Этот шаг требует некоторого планирования с вашей стороны, но он позволяет легко выполнить настройку, когда на экране появятся компоненты. Чтобы добавить компоненты, необходимо создать и настроить объект `GridBagConstraints`. К счастью, один объект может повторно использоваться для всех ваших компонентов, необходимо лишь повторить подготовительную часть перед добавлением каждого элемента. Пример добавления главного компонента — игрового поля:

```
GridBagConstraints gameConstraints = new GridBagConstraints();

gameConstraints.fill = GridBagConstraints.BOTH;
gameConstraints.gridy = 1;
gameConstraints.gridx = 0;
gameConstraints.gridheight = 1;
gameConstraints.gridwidth = 3;

Field field = new Field();
gamePane.add(field, gameConstraints);
```

Обратите внимание: мы указываем, какие ячейки будет занимать поле. В этом заключается суть настройки ограничений `GridBagLayout`. Вы можете отрегулировать такие параметры, как режим заполнения ячеек каждым компонентом и отступы для каждого компонента. Здесь мы просто заполняем все доступное пространство в группе ячеек (`BOTH` — заполнение по горизонтали и по вертикали), но в документации `GridBagLayout` описаны и другие возможности.

Добавим наверху надпись для ведения счета:

```
gameConstraints.fill = GridBagConstraints.BOTH;
gameConstraints.gridy = 0;
gameConstraints.gridx = 0;
gameConstraints.gridheight = 1;
gameConstraints.gridwidth = 1;

JLabel scoreLabel = new JLabel(" Player 1: 0");
gamePane.add(scoreLabel, gameConstraints);
```

Вы заметили, как настройка ограничений напоминает настройку игрового поля? Каждый раз, когда вы сталкиваетесь с подобным сходством, следует подумать об выделении этих действий в функцию, которую можно использовать повторно. Это можно сделать так:

```
private GridBagConstraints buildConstraints(int row, int col,
                                           int rowspan, int colspan)
{
    // Использовать нашу глобальную ссылку на объект gameConstraints
    gameConstraints.fill = GridBagConstraints.BOTH;
    gameConstraints.gridy = row;
    gameConstraints.gridx = col;
    gameConstraints.gridheight = rowspan;
    gameConstraints.gridwidth = colspan;
    return gameConstraints;
}
```

А теперь перепишем прежние блоки кода для надписи и игрового поля:

```
GridBagConstraints gameConstraints = new GridBagConstraints();

JLabel scoreLabel = new JLabel(" Player 1: 0");
Field field = new Field();
gamePane.add(scoreLabel, buildConstraints(0,0,1,1));
gamePane.add(field, buildConstraints(1,0,1,3));
```

При наличии такой функции мы можем быстро добавить другие компоненты и надписи для завершения игрового интерфейса. Например, настройка кнопки броска в правом нижнем углу на рис. 10.27 может выглядеть так:

```
JLabel tossButton = new JButton("Toss");
gamePane.add(tossButton, buildConstraints(2,2,2,1));
```

Намного лучше! Далее вы просто продолжаете создавать компоненты и размещать их в нужных строках и столбцах в соответствующих ячейках. В итоге мы получаем хорошо выглядящий набор компонентов, скомпонованных в одном контейнере.

Как и в других разделах этой главы, мы не сможем рассмотреть все менеджеры макетов — и даже все возможности описанных менеджеров. Обязательно обратитесь к документации Java и попробуйте создать несколько простых приложений для экспериментов с разными макетами. Например, `BoxLayout` неплохо развивает идею сетки, а `GridLayout` удобен для форм ввода данных с красивым выравниванием. А мы пойдем дальше: наконец-то «подключим» все эти компоненты и обеспечим реакцию на ввод текста и нажатия кнопок — все эти действия считаются в Java *событиями*.

События

Если задуматься над архитектурой MVC, мы увидим, что элементы модели и представления просты по своей сути. Мы уже рассмотрели некоторые компоненты Swing, затронули их представления, а также модель более интересных компонентов, таких как `JList`. (Конечно, у надписей и кнопок тоже есть модели, просто они достаточно тривиальны.) С учетом сказанного рассмотрим функциональность контроллера. В Swing (и в Java вообще) взаимодействия между пользователями и компонентами осуществляются посредством событий. Событие содержит общую информацию (например, о том, когда оно произошло), а также информацию, относящуюся к типу события, например координаты точки на экране, в которой был сделан щелчок. *Слушатель (listener)* (или *обработчик (handler)*) выбирает сообщение и может на него отреагировать каким-то полезным образом.

В приведенных ниже примерах можно заметить, что некоторые события и слушатели являются частью пакета `javax.swing.event`, а другие входят в пакет `java.awt.event`. В этом отражен тот факт, что Swing является наследником AWT. Те части AWT, которые все еще актуальны, продолжают использоваться, но в Swing к ним добавился ряд новых элементов, отражающих расширенную функциональность библиотеки.

События мыши

Проще всего начать с примера генерирования и обработки события. Вернемся к приложению `HelloJava` (в новой версии назовем его `HelloMouse`) и добавим слушатель для событий мыши. При щелчке кнопкой мыши это событие будет использоваться для определения позиции `JLabel`. (Кстати говоря, для этого необходимо удалить менеджер макета, потому что мы собираемся задать координаты надписи вручную.) Ниже приведен код нового интерактивного приложения:

```

package ch10;

import java.awt.*;
import javax.swing.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class HelloMouse extends JFrame implements MouseListener { ❶
    JLabel label;

    public HelloMouse() {
        super("MouseEvent Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);
        setSize( 300, 100 );

        label = new JLabel( "Hello, Mouse!", JLabel.CENTER );
        label.setOpaque(true);
        label.setBackground(Color.YELLOW);
        label.setSize(100,20);
        label.setLocation(100,100);
        add(label);

        getContentPane().addMouseListener(this); ❷
    }

    public void mouseClicked(MouseEvent e) { ❸
        label.setLocation(e.getX(), e.getY());
    }

    public void mousePressed(MouseEvent e) { } ❹
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }

    public static void main( String[] args ) {
        HelloMouse frame = new HelloMouse();
        frame.setVisible( true );
    }
}

```

Запустите приложение. Перед вами появляется графический интерфейс в стиле «Hello, World» (рис. 10.28). Надпись будет перемещаться к точке щелчка мышью.



Рис. 10.28. Использование MouseEvent для позиционирования надписи

В исходном коде этого примера следует обратить особое внимание на следующие моменты:

- ❶ При щелчках мыши компьютер генерирует низкоуровневые события, которые передаются в виртуальную машину Java и попадают в ваш код для обработки слушателем. В Java слушатели являются интерфейсами, и вы можете либо создать специальные классы для реализации этих интерфейсов, либо включить все слушатели в основной класс приложения, как сделано в нашем случае. Вопрос о том, где лучше обрабатывать события, зависит от того, что необходимо делать в ответ на них. В оставшейся части книги вы встретите примеры обоих подходов.
- ❷ В дополнение к расширению `JFrame` мы реализовали интерфейс `MouseListener`. Мы должны были предоставить тело каждого метода, входящего в `MouseListener`, но реальная работа выполняется только в `mouseClicked()`. В этом методе мы берем координаты щелчка из объекта события и используем их для изменения позиции надписи. Класс `MouseEvent` содержит разнообразную информацию о событии: когда оно произошло, в каком компоненте оно произошло, какая кнопка мыши была задействована, координаты (x, y) точки щелчка и т. д. Попробуйте вывести часть этой информации в каком-либо из нереализованных методов (например, `mouseDown()`).
- ❸ Обратите внимание, что мы добавили несколько методов для других, неиспользуемых событий мыши. Такая ситуация типична для низкоуровневых событий (таких, как события мыши и клавиатуры). Интерфейсы слушателей спроектированы так, чтобы предоставить вам общую точку для многих взаимосвязанных событий. Вы просто реагируете на конкретные события, которые вас интересуют, а остальные методы оставляете пустыми.
- ❹ Другой критически важный момент в новом коде — вызов `addMouseListener()` для нашей панели содержимого. Синтаксис выглядит немного странно, но этот синтаксис действителен. Использование `getContentPane()` слева означает «здесь события будут генерироваться», а использование `this` как аргумента означает «сюда события будут доставляться». В нашем примере события из панели содержимого окна будут доставляться обратно тому же классу, в котором мы разместили весь код обработки событий мыши.

Адаптеры мыши

Если вы хотите опробовать решение со вспомогательным классом, можно добавить в файл новый, отдельный класс и реализовать `MouseListener` в этом классе. При создании отдельного класса можно воспользоваться сокращением, предоставляемым `Swing` для многих слушателей. Класс `MouseAdapter` — простая пустая реализация интерфейса `MouseListener` с пустыми методами, написанными для каждой разновидности событий. При расширении этого класса вы можете пере-

определить только те методы, которые вас интересуют. Это позволяет создать более понятный и чистый код обработчика.

```
package ch10;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import javax.swing.*;

public class HelloMouseHelper {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "MouseEvent Demo" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        frame.setSize( 300, 300 );

        JLabel label = new JLabel( "Hello, Mouse!", JLabel.CENTER );
        label.setOpaque(true);
        label.setBackground(Color.YELLOW);
        label.setSize(100,20);
        label.setLocation(100,100);
        frame.add(label);

        LabelMover mover = new LabelMover(label);
        frame.getContentPane().addMouseListener(mover);
        frame.setVisible( true );
    }
}

/**
 * Вспомогательный класс для перемещения надписи к позиции щелчка
 */
class LabelMover extends MouseAdapter {
    JLabel labelToMove;
    public LabelMover(JLabel label) {
        labelToMove = label;
    }

    public void mouseClicked(MouseEvent e) {
        labelToMove.setLocation(e.getX(), e.getY());
    }
}
```

О вспомогательных классах важно помнить то, что они должны располагать ссылкой на каждый объект, с которым они взаимодействуют. Заметьте, что мы передали объект надписи конструктору. Это популярный способ установления необходимых связей, но конечно, вы можете добавить необходимую ссылку позднее — главное, чтобы обработчик мог взаимодействовать с каждым необходимым ему объектом до того, как он начнет получать события.

События действий

Хотя события клавиатуры и мыши доступны почти для каждого компонента Swing, их обработка быстро становится однообразной и утомительной. В большинстве GUI-библиотек предусмотрены события высокого уровня, с которыми проще работать. Swing не является исключением. Например, класс `JButton` поддерживает событие `ActionEvent`, которое сообщает о щелчке на кнопке. В большинстве случаев это именно то, что вам нужно. Однако события мыши все еще доступны на случай, если вам нужно более специализированное поведение (например, чтобы реагировать на щелчки разных кнопок мыши или отличать длинное прикосновение от короткого на сенсорном экране).

Для демонстрации обработки событий щелчков часто используется простой счетчик вроде изображенного на рис. 10.29. Каждый раз, когда вы щелкаете кнопкой мыши, надпись обновляется. Этот простой пример показывает, что вы можете добавить в приложение несколько кнопок с разными реакциями. Посмотрим, как выполняется связывание в этом приложении:

```
package ch10;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ActionDemo1 extends JFrame implements ActionListener {
    int counterValue = 0;
    JLabel counterLabel;

    public ActionDemo1() {
        super( "ActionEvent Counter Demo" );
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        setSize( 300, 180 );

        counterLabel = new JLabel( "Count: 0", JLabel.CENTER );
        add(counterLabel);

        JButton incrementer = new JButton("Increment");
        incrementer.addActionListener(this);
        add(incrementer);
    }

    public void actionPerformed(ActionEvent e) {
        counterValue++;
        counterLabel.setText("Count: " + counterValue);
    }

    public static void main( String[] args ) {
        ActionDemo1 demo = new ActionDemo1();
        demo.setVisible(true);
    }
}
```

Мы обновляем простую переменную-счетчик и выводим результат внутри метода `actionPerformed()`, в котором объекты `ActionListener` получают свои события. Здесь используется подход с непосредственной реализацией слушателя, но с таким же успехом можно было создать вспомогательный класс, как это было сделано в первом примере из раздела «События мыши», с. 365.

События действий относительно просты; они не содержат такой подробной информации, как события мыши, зато содержат свойство `command`. Это свойство может настраиваться, но для кнопок по умолчанию в нем передается текст на кнопке. Класс `TextField` также генерирует действие при нажатии клавиши Enter (Return) во время ввода в текстовом поле. В этом случае в свойстве `command` передается текст, в настоящее время содержащийся в поле. На рис. 10.30 показано небольшое приложение, которое связывает кнопку и текстовое поле с надписью.



Рис. 10.29. Использование `ActionEvent` для увеличения счетчика



Рис. 10.30. Использование событий `ActionEvent` от разных источников

```
public class ActionDemo2 {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "ActionListener Demo" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());
        frame.setSize( 300, 180 );

        JLabel label = new JLabel( "Results go here", JLabel.CENTER );
        ActionCommandHelper helper = new ActionCommandHelper(label);

        JButton simpleButton = new JButton("Button");
        simpleButton.addActionListener(helper);

        JTextField simpleField = new JTextField(10);
        simpleField.addActionListener(helper);

        frame.add(simpleButton);
        frame.add(simpleField);
        frame.add(label);
    }
}
```

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

/**
 * Вспомогательный класс для вывода свойства command любого события
 * ActionEvent в заданной надписи
 */
class ActionCommandHelper implements ActionListener {
    JLabel resultLabel;

    public ActionCommandHelper(JLabel label) {
        resultLabel = label;
    }

    public void actionPerformed(ActionEvent ae) {
        resultLabel.setText(ae.getActionCommand());
    }
}
```

Обратите внимание на интересную особенность этого кода: один объект `ActionListener` служит нам для обработки событий как кнопки, так и текстового поля! Это большое достоинство той системы слушателей, которая используется в Swing при обработке событий. Любой компонент, генерирующий событие заданного типа, может оповещать любой слушатель, прослушивающий события данного типа. Иногда обработчики событий уникальны, и тогда вы должны написать отдельный обработчик для каждого компонента. Но во многих приложениях одна и та же операция может выполняться несколькими способами. Часто эти разные варианты ввода удается обрабатывать одним слушателем. А чем меньше объем кода, тем меньше потенциальных проблем!

События изменений

Другой тип событий, встречающийся в нескольких компонентах Swing, — `ChangeEvent`. Это простые события, которые позволяют узнать о неких изменениях. Например, класс `JSlider` использует этот механизм для передачи информации об изменениях в позиции ползунка. Класс `ChangeEvent` располагает ссылкой на изменившийся компонент (*источник события*), но не имеет подробной информации о том, что могло измениться в этом компоненте. Вы сами должны обратиться к компоненту за подробностями. Этот процесс по принципу «прослушивать, затем запросить» может показаться утомительным, но он позволяет организовать эффективные уведомления о необходимости обновлений без создания сотен классов с тысячами методов для всех потенциальных комбинаций.

Мы не будем воспроизводить здесь все приложение, а ограничимся фрагментом, показывающим, как наша игра использует `ChangeListener` для связывания ползунка прицеливания с физиком:

```

gamePane.add(buildAngleControl(), buildConstraints(2, 0, 1, 1));
// ...

private JSlider buildAngleControl() {
    // Угол прицела лежит в диапазоне от 0 до 180 градусов
    JSlider slider = new JSlider(0,180);

    // Но тригонометрический  $\theta$  находится справа, а не слева
    slider.setInverted(true);

    // И теперь при каждом изменении значения ползунка
    // необходимо выполнять обновление
    slider.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
            player1.setAimingAngle((float)slider.getValue());
            field.repaint();
        }
    });
    return slider;
}

```

В этом фрагменте мы взяли фабричный паттерн, чтобы создать ползунок и вернуть его для использования в методе `add()` контейнера `gamePane`. Как видите, здесь создается простой анонимный внутренний класс. Изменение ползунка имеет только один эффект и является единственным способом прицеливания. Поскольку нет возможности повторного использования класса, анонимный внутренний класс здесь очень эффективен. Нет ничего плохого в том, чтобы создать полный вспомогательный класс и передать ему элементы `player1` и `field` в аргументах конструктора или метода инициализации, но представленный выше подход часто встречается на практике. Хотя на первый взгляд он выглядит немного странно, все станет просто, когда вы освоите этот паттерн. Он является самодокументируемым, и вы можете быть уверены в отсутствии скрытых побочных эффектов. Для программиста ситуация «что видишь, то и получаешь» идеальна.

Наш `Widget` не совсем хорошо подходит для экспериментов с событиями и исправлением ошибок в `jshell`. Конечно, вы можете ввести в командной строке даже такой код, как анонимный внутренний класс `ChangeListener`, но это утомительно и ненадежно: командная строка затрудняет исправление ошибок. Как правило, проще писать маленькие специализированные демоприложения. Хотя мы рекомендуем вам запустить игру и поэкспериментировать с ползунком из приведенного выше кода, но также вам надо попробовать свои силы в написании собственных приложений.

Другие события

Существуют десятки других событий и слушателей, распределенных по пакетам `java.awt.event` и `javax.swing.event`. Вам стоит заглянуть в документацию

просто для того, чтобы получить представление о других типах событий, с которыми вы можете столкнуться. В табл. 10.2 перечислены события и слушатели, связанные с компонентами, которые рассмотрены ранее в этой главе, а также другие, заслуживающие вашего внимания в ходе работы со Swing. Еще раз подчеркнем, что этот список неполон, но он поможет вам работать с этими базовыми компонентами и позволит обрести уверенность для самостоятельного изучения других компонентов и их событий.

Таблица 10.2. События Swing / AWT и связанные с ними слушатели

S/A	Класс события	Интерфейс слушателя	Генерирующие компоненты
A	ActionEvent	ActionListener	JButton, JMenuItem, JTextField
S	ChangeEvent	ChangeListener	JSlider
A	ItemEvent	ItemListener	JCheckBox, JRadioButton
A	KeyEvent	KeyListener	Потомки Component
S	ListSelectionEvent	ListSelectionListener	JList
A	MouseEvent	MouseListener	Потомки Component
A	MouseMotionEvent	MouseMotionListener	Потомки Component
События AWT (A) — из пакета java.awt.event, события Swing (S) — из пакета javax.swing.event			

Если вы не уверены в том, какие события поддерживает тот или иной компонент, обратитесь к его документации и поищите методы с именами вида `addXYZListener()`. Тип **XYZ** прямо указывает на то, где продолжить поиски в документации. Когда вы найдете документацию по слушателю, попробуйте реализовать каждый метод и просто вывести информацию о полученном событии. Экспериментируя, ошибаясь и исправляя ошибки, вы многое узнаете о том, как разные компоненты Swing реагируют на события клавиатуры и мыши.

Модальные и всплывающие окна

События позволяют пользователю привлечь ваше внимание. Точнее, привлечь внимание какого-то метода к тому, что делает пользователь. А как быть, когда вам нужно привлечь внимание пользователя? Для этого нужны *всплывающие окна* (*pop-up windows*). Их также называют «модальными», «диалоговыми» и даже «модальными диалоговыми». Название «диалоговое окно» происходит от того факта, что оно сообщает пользователю некоторую информацию и ожидает (или требует) ответа. Диалог получается не самый многословный, но все же.

Название «модальное окно» означает, что некоторые из окон, требующих ответа, фактически блокируют дальнейшую работу приложения, пока не получат ответ. Вероятно, такие окна встречались вам во многих десктопных приложениях. Например, если приложение должно обновиться, то при запуске оно может вывести модальное диалоговое окно с кнопкой для запуска процесса обновления. Приложение ставит вас перед тем фактом, что перешло в ограниченный режим до тех пор, пока вы не сообщите ему, как следует продолжать.

«Всплывающее окно» — более общий термин. Хотя всплывающие окна могут быть модальными, вы также можете создавать простые («немодальные», хотя это техническое определение постепенно исчезает) всплывающие окна, которые не препятствуют работе со всеми остальными частями приложения. Представьте диалоговое окно поиска, которое можно оставить на экране и вернуться к основному документу в текстовом редакторе.

Диалоговые окна сообщений

Swing предоставляет минимальный класс `JDialog`, предназначенный для создания специализированных диалоговых окон. Но для типичных диалоговых взаимодействий с пользователями класс `JOptionPane` предоставляет ряд более удобных лаконичных окон.

Пожалуй, наибольшее раздражение вызывает диалоговое окно в стиле «что-то пошло не так». Оно не сообщает вам ничего, кроме того, что приложение работает не так, как ожидалось. Обычно такое окно содержит короткое сообщение и кнопку **OK**, нажатие которой закрывает окно. Такие окна создаются для того, чтобы приостановить работу программы до тех пор, пока пользователь не подтвердит, что он увидел сообщение. На рис. 10.31 приведен простой пример вывода диалогового окна сообщения по нажатию кнопки.



Рис. 10.31. Простое модальное окно на базе `JOptionPane`

```
package ch10;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ModalDemo extends JFrame implements ActionListener {

    JLabel modallabel;

    public ModalDemo() {
        super( "Modal Dialog Demo" );
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        setSize( 300, 180 );

        modallabel = new JLabel( "Press 'Go' to show the popup!", JLabel.CENTER );
        add(modallabel);

        JButton goButton = new JButton("Go");
        goButton.addActionListener(this);
        add(goButton);
    }

    public void actionPerformed(ActionEvent ae) {
        JOptionPane.showMessageDialog(this, "We're going!", "Alert",
            JOptionPane.INFORMATION_MESSAGE);
        modallabel.setText("Go pressed! Press again if you like.");
    }

    public static void main(String args[]) {
        ModalDemo demo = new ModalDemo();
        demo.setVisible(true);
    }
}
```

Вероятно, вы заметили код, связывающий кнопку `goButton` со слушателем `this`. Это тот же паттерн, который использовался с первым событием `ActionEvent`. Впрочем, изменилась обработка события. Мы выводим диалоговое окно сообщения, а затем обновляем надпись, сообщая об успешном отображении диалогового окна.

Вызов `showMessageDialog()` получает четыре аргумента. Аргумент `this` в первой позиции определяет фрейм или окно, которое является «владельцем» всплывающего окна; при этом оповещение пытается выровняться по центру владельца. В качестве владельца указывается само приложение. Во втором и третьем аргументе передаются, соответственно, строки с текстом сообщения и заголовком окна. Последний аргумент обозначает тип всплывающего окна; в основном он

влияет на значок, который будет отображаться. Поддерживаются следующие типы:

- `ERROR_MESSAGE` — с красным значком «Стоп».
- `INFORMATION_MESSAGE` — со значком Дюка¹.
- `WARNING_MESSAGE` — с желтым треугольником.
- `QUESTION_MESSAGE` — со значком Дюка.
- `PLAIN_MESSAGE` — без значка.

Если вы хотите поэкспериментировать с этими всплывающими окнами, вернитесь к `jshell`. Используйте в качестве владельца наш объект `Widget` или удобное значение `null`, которое указывает, что конкретного фрейма или окна нет, но всплывающее окно должно приостановить всю работу приложения и появиться в центре экрана:

```
jshell> import javax.swing.*  
  
jshell> JOptionPane.showMessageDialog(null, "Hi there", "jshell Alert",  
JOptionPane.ERROR_MESSAGE)
```



Рис. 10.32. Окно `JOptionPane`, открытое из `jshell`

Возможно, вам придется запустить `ModalDemo` несколько раз; следите за текстом в объекте `modalLabel`. Обратите внимание: он изменяется только **после** того, как вы закроете всплывающее окно. Важно помнить, что модальные диалоговые окна прерывают нормальную последовательность выполнения приложения. Именно такие окна нужны для оповещений об ошибках или для получения обязательного ввода от пользователя, но для простых обновлений статуса они вряд ли подойдут.

Нетрудно придумать и другие, более полезные применения для таких оповещений. А если вы столкнетесь с ситуацией «что-то сломалось» в своем приложении,

¹ Дюк — официальный талисман Java (<https://ru.wikipedia.org/wiki/Java>).

постарайтесь предоставить пользователю осмысленное сообщение об ошибке, которое поможет ему справиться с ситуацией. Вспомните регулярное выражение для проверки адресов электронной почты из раздела «Класс Pattern», с. 281. Можно присоединить `ActionListener` к текстовому полю, и когда пользователь нажмет Enter (Return), вывести диалоговое окно с сообщением об ошибке, если содержимое поля не похоже на адрес электронной почты.

Диалоговые окна подтверждения

Другое распространенное применение для всплывающих окон — проверка намерений пользователя. Многие приложения спрашивают, действительно ли пользователь хочет завершить работу, что-то удалить или совершить другое необратимое действие (например, обрести силу «Камней бесконечности»). `JOptionPane` придет вам на помощь. Новое диалоговое окно можно опробовать в `jshell`:

```
jshell> JOptionPane.showConfirmDialog(null, "Are you sure?")
$18 ==> 0
```



Рис. 10.33. `JOptionPane` с подтверждением

На экране должно появиться всплывающее окно с кнопками `Yes`, `No` и `Cancel` (рис. 10.33). Вы можете определить, какой ответ был выбран пользователем, по возвращаемому значению (`int`) метода `showConfirmDialog()`. (Запустив этот пример во время написания главы, мы нажали кнопку `Yes`. Ей соответствует возвращаемое значение `0` в приведенном выше фрагменте `jshell`.) Изменим вызов для обработки этого ответа и снова нажмем кнопку `Yes`:

```
jshell> int answer = JOptionPane.showConfirmDialog(null, "Are you sure?")
answer ==> 0

jshell> answer == JOptionPane.YES_OPTION ? "They said yes!" :
" They said no or canceled. :("
$20 ==> "They said yes!"
```

Существуют и другие стандартные диалоговые окна подтверждения, которые можно вывести при помощи двух дополнительных аргументов. Первый аргумент — строка `String` с заголовком диалогового окна, второй — один из следующих типов:

- `YES_NO_OPTION`
- `YES_NO_CANCEL_OPTION`
- `OK_CANCEL_OPTION`

В нашем примере дополнительные аргументы не указаны, поэтому мы получили заголовок по умолчанию «Select an Option» и набор кнопок, определяемых константой `YES_NO_CANCEL_OPTION`. Обычно одновременное присутствие кнопок `No` и `Cancel` только путает пользователя. Мы советуем использовать сочетания: «Yes и No», или «OK и Cancel», или только «OK», но не «Yes, No и Cancel». Пользователь всегда может закрыть диалоговое окно стандартной угловой кнопкой X без выбора какой-либо кнопки с надписью. Чтобы выявить эту ситуацию, проверьте результат на условие `JOptionPane.CLOSED_OPTION`.

Здесь эта тема рассматриваться не будет, однако вы можете использовать метод `showOptionDialog()` для создания аналогичных окон подтверждения, но с нестандартными наборами кнопок. Как обычно, вам поможет в этом документация JDK!

Диалоговые окна ввода

Последняя разновидность всплывающих окон — окна, которые запрашивают короткий произвольный ввод. Метод `showInputDialog()` позволяет задать пользователю вопрос, на который он вводит ответ. Этот ответ (`String`) можно сохранить по аналогии с сохранением выбранного варианта при подтверждении. Добавим еще одну кнопку открытия всплывающего окна в своем демоприложении (рис. 10.34):



Рис. 10.34. Использование `JOptionPane` для ввода

```
jshell> String pin = JOptionPane.showInputDialog(null, "Please enter your PIN:")
pin ==> "1234"
```

Такой способ хорошо подходит для одноразовых запросов, но не для серий вопросов, которые вы хотите задавать пользователю. Выводить модальные окна надо только для кратковременных операций, так как эти окна прерывают работу пользователя. Иногда это именно то, что нужно. Но если вы начнете злоупотреблять его вниманием, то он, скорее всего, начнет раздражаться и игнорировать все всплывающие окна в вашем приложении.

Влияние многопоточности

Если вы читали документацию Swing из JDK во время работы над этой главой, то вам, возможно, встретилось предупреждение о том, что компоненты Swing не являются потокобезопасными. Как говорилось в главе 9, Java поддерживает многопоточность для использования вычислительной мощности современных компьютеров. Одна из проблем многопоточных приложений заключается в том, что два потока могут конкурировать за один ресурс или обновлять одну переменную одновременно, но разными значениями. А если вы не уверены в правильности своих данных, то код может стать трудным в отладке и некорректно работать. Применительно к компонентам Swing это предупреждение в документации напоминает, что ваши GUI-элементы подвержены потенциальным проблемам такого рода.

Как же добиться нормальной работы пользовательских интерфейсов? Для этого следует обновлять компоненты Swing *из потока диспетчеризации событий* AWT. Этот поток обрабатывает такие события, как, например, щелчки мышью. Если компонент обновляется в ответ на событие (как в примере со счетчиком в разделе «События действий», с. 369), то проблема решена. Идея такова: если все остальные потоки вашего приложения отправляют обновления пользовательского интерфейса в один общий поток диспетчеризации событий, то ни один компонент не может быть поврежден из-за одновременных (возможно, конфликтующих) воздействий на него.

Типичный пример ситуации, в которой многопоточность выходит на первый план в графических приложениях, — так называемые длительные операции. Представьте, что вы загружаете файл из облачного хранилища, а на экране в это время крутится анимированный спиннер, который должен вас развлекать. А если вам надоело ждать? Если вы подозреваете, что загрузка прервалась, но спиннер все еще крутится? Если ваша длительная операция использует поток диспетчеризации событий, то пользователь не сможет нажать кнопку отмены и вообще выполнить какие-либо действия. Длительные операции должны обрабатываться в отдельных потоках, выполняемых в фоновом режиме, чтобы ваше приложение оставалось доступным и реагировало на действия пользователя. Но как обновить пользовательский интерфейс при завершении фонового потока? В Swing существует помощник специально для этой задачи.

SwingUtilities и обновление компонентов

Вы можете использовать класс `SwingUtilities` из любого потока, чтобы безопасно и стабильно обновлять GUI-компоненты. Для взаимодействия с GUI предназначены два статических метода:

- `invokeAndWait()`
- `invokeLater()`

Как подсказывают эти имена, первый метод выполняет некий код обновления пользовательского интерфейса и заставляет текущий поток ожидать завершения этого кода, прежде чем продолжать выполнение. Второй метод передает код обновления пользовательского интерфейса потоку диспетчеризации событий, после чего немедленно возобновляет выполнение в текущем потоке. Какой из двух методов выбрать? Это зависит от того, должен ли ваш фоновый поток знать состояние пользовательского интерфейса перед продолжением. Например, если вы добавляете новую кнопку в свой интерфейс, лучше использовать `invokeAndWait()`. Тогда к моменту продолжения фоновый поток будет точно знать, что при последующем обновлении добавленной кнопки она будет существовать.

А если вас не особенно интересует, когда произойдет предполагаемое обновление? Если важно лишь то, что оно в итоге будет безопасно обработано потоком диспетчеризации? Тогда идеально подойдет `invokeLater()`. Представьте себе обновление индикатора длительного процесса загрузки большого файла. Ничто не мешает вам инициировать сразу несколько обновлений по мере того, как загрузка приближается к завершению. Вам не нужно дожидаться, пока эти графические обновления завершатся, чтобы продолжать загрузку. Если очередное обновление такого индикатора будет немного отложено или сработает совсем незадолго до следующего обновления, то ничего плохого не случится. Зато вам надо предусмотреть, чтобы загрузка файла не прервалась из-за сильно загруженного работой графического интерфейса, особенно если сервер чувствителен к паузам.

Несколько примеров подобных взаимодействий между сетью и пользовательским интерфейсом мы рассмотрим в следующей главе. А пока давайте смоделируем сетевой трафик с обновлением маленькой надписи, чтобы показать работу `SwingUtilities`. Кнопка `Start` обновляет надпись значением в процентах и запускает фоновый поток, который просто ожидает в течение секунды, а затем увеличивает это значение. Каждый раз, когда поток активизируется, он обновляет надпись с помощью метода `invokeLater()` для формирования текста надписи. Вот как это делается:

```
public class ProgressDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "SwingUtilities 'invoke' Demo" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());
        frame.setSize( 300, 180 );

        JLabel label = new JLabel("Download Progress Goes Here!",
                                   JLabel.CENTER );
        Thread pretender = new Thread(new ProgressPretender(label));

        JButton simpleButton = new JButton("Start");
        simpleButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                simpleButton.setEnabled(false);
                pretender.start();
            }
        });

        JLabel checkLabel = new JLabel("Can you still type?");
        JTextField checkField = new JTextField(10);

        frame.add(label);
        frame.add(simpleButton);
        frame.add(checkLabel);
        frame.add(checkField);
        frame.setVisible(true);
    }
}
```

Основная часть этого кода должна быть вам знакома, но мы хотим подчеркнуть несколько интересных деталей. Прежде всего взгляните на то, как мы создаем поток. В аргументе конструктора `Thread` передается вызов `new ProgressPretender`. В принципе, можно было бы разделить вызовы, но мы не собираемся снова обращаться к объекту `ProgressPretender`, поэтому можем выбрать более компактный и элегантный подход. Впрочем, к самому потоку мы все-таки **будем** обращаться, поэтому для него создается нормальная переменная. После этого поток запускается в `ActionListener` для нашей кнопки. Обратите внимание на блокирование кнопки `Start` в этом слушателе. Пользователь не должен запускать уже работающий поток!

Также обратите внимание на добавленное текстовое поле. Пока индикатор обновляется, приложение должно нормально реагировать на действия пользователя (например, на ввод текста). Попробуйте сами! Конечно, текстовое поле ни с чем не связано, но текст должен нормально вводиться и удаляться, пока значение счетчика медленно ползет вверх (рис. 10.35).

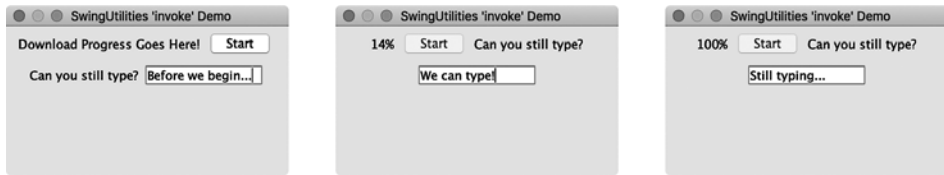


Рис. 10.35. Потокбезопасные обновления индикатора прогресса

Как же нам удалось обновлять надпись, не блокируя приложение? Посмотрим на класс `ProgressPretender` и проанализируем метод `run()`:

```
class ProgressPretender implements Runnable {
    JLabel label;
    int progress;

    public ProgressPretender(JLabel label) {
        this.label = label;
        progress = 0;
    }

    public void run() {
        while (progress <= 100) {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    label.setText(progress + "%");
                }
            });
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                System.err.println("Someone interrupted us. Skipping download.");
                break;
            }
            progress++;
        }
    }
}
```

В этом классе сохраняется объект надписи, переданный конструктору, чтобы мы знали, где следует выводить обновленное значение индикатора прогресса. Метод `run()` состоит из трех основных шагов: 1) обновление надписи; 2) приостановка на 1000 миллисекунд; 3) увеличение значения индикатора.

Что касается шага 1, обратите внимание на довольно сложный аргумент, передаваемый методу `invokeLater()`. Он похож на определение класса, но основан на интерфейсе `Runnable`, который рассматривался в главе 9. Это еще один пример использования анонимных внутренних классов в Java. Существуют и другие способы создания объектов `Runnable`, но этот потоковый паттерн очень популярен, как и обработка простых событий с помощью анонимных слушателей.

Вложенный аргумент `Runnable` обновляет надпись текущим значением, но еще раз подчеркнем, что обновление выполняется в потоке диспетчеризации событий. Именно это «волшебство» сохраняет нормальную реакцию текстового поля, хотя наш «поток прогресса» бездействует большую часть времени.

Шаг 2 — стандартная приостановка потока. Вспомните, что метод `sleep()` знает, что он может быть прерван, поэтому компилятор следит за тем, чтобы вы предоставили блок `try / catch`, как было сделано выше. Есть много способов обработки прерывания, но в данном случае мы решили просто прервать цикл.

Наконец, на шаге 3 значение счетчика увеличивается, а весь процесс начинается заново. После достижения 100 цикл завершается и надпись-индикатор перестает меняться. Если запасетесь терпением и подождете, вы увидите это итоговое значение. При этом само приложение останется активным: вы по-прежнему сможете вводить текст в текстовом поле. «Загрузка» завершена, все в порядке!

Таймеры

В библиотеке Swing есть таймер, предназначенный для работы в пользовательских интерфейсах. Класс `javax.swing.Timer` достаточно прост. Он ожидает в течение заданного периода времени, а затем выдает событие действия — однократно или многократно. В графических приложениях бывает много ситуаций, в которых уместны таймеры. Помимо циклов анимации, они могут автоматически отменять некоторые действия, например загрузку с сетевого ресурса, если она занимает слишком много времени. А еще можно вывести спиннер или сообщение «Пожалуйста, подождите», чтобы пользователь знал, что операция все еще выполняется... Или можно убрать модальное диалоговое окно, если пользователь не ответил за заданный промежуток времени. Во всех ситуациях такого рода отлично подойдет простой одноразовый таймер.

Анимация на базе `Timer`

Вернемся к анимации летящих яблок из раздела «Снова об анимации», с. 310, и попробуем осуществить ее с помощью экземпляра `Timer`. На самом деле мы обошли вопрос использования правильного вспомогательного метода, такого как `invokeLater()`, для безопасной перерисовки игры при использовании стандартных потоков. Класс `Timer` позаботится об этой важной детали за нас. К счастью, мы все еще можем использовать метод `step()` класса `Apple` из первой версии анимации. Нужно только изменить стартовый метод и сохранить переменную для объекта таймера:

```
public static final int STEP = 40; // Продолжительность кадра в миллисекундах
Timer animationTimer;

// ...
```

```

void startAnimation() {
    if (animationTimer == null) {
        animationTimer = new Timer(STEP, this);
        animationTimer.setActionCommand("repaint");
        animationTimer.setRepeats(true);
        animationTimer.start();
    } else if (!animationTimer.isRunning()) {
        animationTimer.restart();
    }
}

// ...

public void actionPerformed(ActionEvent event) {
    if (animating && event.getActionCommand().equals("repaint")) {
        System.out.println("Timer stepping " + apples.size() + " apples");
        for (Apple a : apples) {
            a.step();
            detectCollisions(a);
        }
        repaint();
        cullFallenApples();
    }
}

```

У такого подхода есть два достоинства. Код явно проще читается, потому что мы не отвечаем за паузы между действиями. При создании `Timer` конструктору передается интервал между событиями и объект `ActionListener` для получения событий — в данном случае наш класс `Field`. Мы предоставляем таймеру команду действия, устанавливаем его на повторение и запускаем! Как было замечено при перечислении причин для использования таймеров, другое достоинство относится к специфике Swing и графических приложений: `javax.swing.Timer` выдает свои события действий *в потоке диспетчеризации событий*. Вам не нужно ничего упаковывать в `invokeAndWait()` или `invokeLater()`. Просто включите код, который должен выполняться по таймеру, в метод `actionPerformed()` присоединенного слушателя — и все готово!

Так как объекты `ActionEvent` могут генерироваться несколькими компонентами, мы приняли меры к предотвращению возможных коллизий присваиванием атрибута `actionCommand` для нашего таймера. В данном случае этот шаг не является строго необходимым, но он оставляет классу `Field` возможность попутной обработки других событий без нарушения анимации.

Другие применения `Timer`

В качественном, хорошо продуманном приложении возникает немало мелких моментов, в которых было бы полезно воспользоваться одноразовым таймером. Наша игра с яблоками очень проста по сравнению с большинством коммерческих

приложений и игр, но даже здесь таймер позволит добавить немного «реализма»: после броска физик делает короткую паузу, прежде чем сможет бросить следующее яблоко. Будем считать, что он должен нагнуться и взять другое яблоко перед тем, как прицелиться для нового броска. Подобные задержки хорошо реализуются с помощью таймера.

Мы добавим паузу в тот фрагмент кода класса `Field`, в котором происходит бросок:

```
public void startTossFromPlayer(Physicist physicist) {
    if (!animating) {
        System.out.println("Starting animation!");
        animating = true;
        startAnimation();
    }

    if (animating) {
        // Проверить, что у нас есть яблоко для броска
        if (physicist.aimingApple != null) {
            Apple apple = physicist.takeApple();
            apple.toss(physicist.aimingAngle, physicist.aimingForce);
            apples.add(apple);
            Timer appleLoader = new Timer(800, physicist);
            appleLoader.setActionCommand("New Apple");
            appleLoader.setRepeats(false);
            appleLoader.start();
        }
    }
}
```

На этот раз таймер устанавливается на однократное срабатывание вызовом `setRepeats(false)`. Это означает, что менее чем через секунду физику будет отправлено одиночное событие. В свою очередь, в определении класса `Physicist` необходимо добавить секцию `implements ActionListener` и соответствующую функцию `actionPerformed()`:

```
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("New Apple")) {
        getNewApple();
        if (field != null) {
            field.repaint();
        }
    }
}
```

Подобные задачи можно решать и без `Timer`, но в Swing сочетание эффективных событий с отсчетом времени и автоматического использования потока диспетчеризации событий делает этот класс достойным кандидатом. На крайний случай

он отлично подойдет для прототипов приложения. При необходимости вы всегда можете вернуться и провести рефакторинг приложения, чтобы использовать специализированный многопоточный код.

Что дальше?

Как было сказано в начале этой главы, мир графических приложений велик, и в нем найдется множество тем и вопросов, которые вам придется рассматривать самостоятельно. Мы покажем лишь ряд ключевых тем, на которых вам надо сосредоточиться в первую очередь, если у вас есть планы разработки десктопных приложений.

Меню

Хотя это и не является формальным требованием, в большинстве десктопных приложений есть меню базовых операций (например, для сохранения измененных файлов или настройки конфигурации) и меню специфических операций (например, для сортировки данных по столбцу или выделенному участку в таблицах). Классы `JMenu`, `JMenuBar` и `JMenuItem` помогают добавить эту функциональность в Swing-приложения. Все меню открываются из строки меню и состоят из отдельных команд. Swing содержит три готовых класса для команд меню: `JMenuItem` для базовых команд, `JCheckboxMenuItem` для команд-флажков с двумя состояниями и `JRadioButtonMenuItem` для группируемых команд меню (например, для выбора шрифта или цветовой темы). Класс `JMenu` сам по себе является действительной командой меню, что позволяет строить системы вложенных меню. `JMenuItem` своим поведением напоминает кнопку (как и родственные классы команд меню), и для перехвата событий меню используются те же слушатели.

На рис. 10.36 показан пример простой строки меню, содержащей меню и команды.

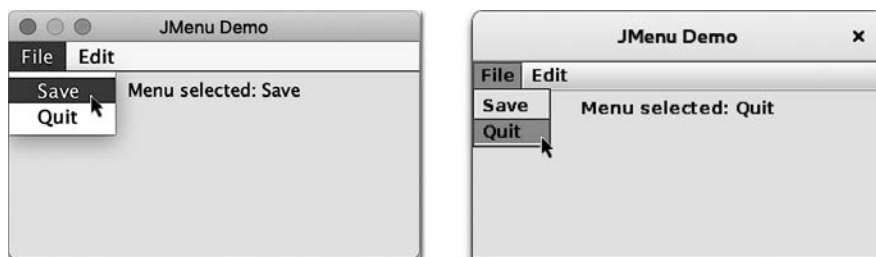


Рис. 10.36. `JMenu` и `JMenuItem` в macOS и Linux

Обратите внимание: приложения для macOS и Linux выглядят немного по-разному. Swing и Java отражают многие особенности тех платформ, на которых работают. Впрочем, в данном случае есть явное несоответствие традиции: главное меню в приложениях macOS принято располагать в самой верхней строке главного окна. Когда у вас появится опыт программирования и вы захотите расширять свои приложения или передавать свой код другим разработчикам, вам понадобится выполнять платформенно-зависимые операции (например, использовать традиционное меню macOS и графические значки для запуска приложений). А пока обойдемся более простым вариантом меню:

```
package ch10;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MenuDemo extends JFrame implements ActionListener {
    JLabel resultsLabel;

    public MenuDemo() {
        super( "JMenu Demo" );
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        setSize( 300, 180 );

        resultsLabel = new JLabel("Click a menu item!" );
        add(resultsLabel);

        // Теперь создадим два меню и заполним их
        JMenu fileMenu = new JMenu("File");
        JMenuItem saveItem = new JMenuItem("Save");
        saveItem.addActionListener(this);
        fileMenu.add(saveItem);
        JMenuItem quitItem = new JMenuItem("Quit");
        quitItem.addActionListener(this);
        fileMenu.add(quitItem);

        JMenu editMenu = new JMenu("Edit");
        JMenuItem cutItem = new JMenuItem("Cut");
        cutItem.addActionListener(this);
        editMenu.add(cutItem);
        JMenuItem copyItem = new JMenuItem("Copy");
        copyItem.addActionListener(this);
        editMenu.add(copyItem);
        JMenuItem pasteItem = new JMenuItem("Paste");
        pasteItem.addActionListener(this);
        editMenu.add(pasteItem);

        // И наконец, создадим JMenuBar для приложения
        JMenuBar mainBar = new JMenuBar();
        mainBar.add(fileMenu);
        mainBar.add(editMenu);
    }
}
```

```

        setJMenuBar(mainBar);
    }

    public void actionPerformed(ActionEvent event) {
        resultsLabel.setText("Menu selected: " + event.getActionCommand());
    }

    public static void main(String args[]) {
        MenuDemo demo = new MenuDemo();
        demo.setVisible(true);
    }
}

```

Очевидно, в данном случае мы не можем сделать ничего содержательного с действиями команд меню, но мы хотим показать, как начать создание тех частей, которые должны присутствовать в каждом профессиональном приложении.

Хранение конфигураций

API конфигураций (Java Preferences API) позволит вам длительно хранить все системные и пользовательские настройки вашего приложения в периоды между его запусками в виртуальной машине Java. API конфигураций напоминает переносимую версию реестра Windows — это компактная база данных, в которой сосредоточены небольшие записи, доступные всем приложениям. Записи хранятся в форме пар «имя — значение», где значения могут относиться к одному из нескольких стандартных типов, включая строки, числа, логические значения и даже короткие байтовые массивы (помните: речь идет о **небольших** объемах данных). Когда вы начнете писать полноценные десктопные приложения, в них наверняка будут какие-то элементы, настраиваемые пользователями. API конфигураций предоставляет удобный механизм для хранения такой информации в кросс-платформенном формате, с которым удобно работать. Это делает приложения более привлекательными для пользователей.

За дополнительной информацией обращайтесь к документации Oracle (<https://docs.oracle.com/javase/8/docs/technotes/guides/preferences/index.html>).

Нестандартные компоненты и Java 2D

Мы кратко затронули тему создания нестандартных компонентов в нашей игре, а также ее класса `Field`. Мы предоставили собственный метод `paintComponent()` для прорисовки объектов игры — яблок, деревьев и физиков. Это было только началом, и теперь вы можете **значительно** расширить функциональность этого приложения. Например, возьмите низкоуровневые события мыши и клавиатуры и попробуйте связать их с более эффектным визуальным интерфейсом. Попробуйте генерировать свои нестандартные события. Вы можете написать

свой собственный менеджер макета и даже нарисовать совершенно новый графический стиль, который распространится на все компоненты библиотеки Swing! Эта потрясающая расширяемость требует достаточно глубоких знаний о Swing и Java, но она существует и ждет вас.

Что касается рисования, для начала ознакомьтесь с Java 2D API в документации Oracle (<https://docs.oracle.com/javase/8/docs/technotes/guides/2d/>). Этот API предоставляет ряд приятных обновлений для графических средств пакета AWT. Если вас все-ррез заинтересуют возможности 2D-графики Java, прочтите книгу Джонатана Кнудсена (Jonathan Knudsen) «Java 2D Graphics». Уже упоминавшаяся книга Лоя (Loy) и соавторов «Java Swing» (2-е издание) станет хорошим источником информации обо всем, что касается Swing.

JavaFX

Еще один API, заслуживающий вашего внимания, — JavaFX. Эта подборка пакетов изначально проектировалась для замены Swing; она включает расширенные медийные средства (в том числе для работы с видео и высококачественным звуком). JavaFX настолько отличается от Swing, что они вряд ли смогут сосуществовать в составе JDK. Похоже, у Oracle нет планов отказаться от Swing или объявить его устаревшим. При Java 11 (напомним, что это текущая версия с долгосрочной поддержкой) в OpenJDK появилась поддержка JavaFX в форме проекта OpenJFX. Дополнительную информацию можно найти по адресу <https://openjfx.io>.

Думайте о пользователях

В этой главе был представлен краткий обзор основных элементов, которые понадобятся вам для создания пользовательских интерфейсов десктопных приложений. Были рассмотрены такие компоненты, как `JButton`, `JLabel` и `JTextField`, они почти наверняка будут в любом графическом приложении, которое вы пишете. Мы рассказали о размещении этих компонентов в контейнерах, а также о создании более сложных комбинаций контейнеров и компонентов для формирования более интересных интерфейсов. Надеемся, что мы также показали вам достаточное количество дополнительных компонентов, благодаря которым ваши приложения понравятся пользователям.

Но сами по себе десктопные приложения — это еще не все, что нужно пользователям. В наши дни такие приложения, как правило, должны взаимодействовать через интернет с другим программным обеспечением. В двух последующих главах мы расскажем об основах программирования сетевых коммуникаций и веб-приложений.

Сетевые коммуникации и ВВОД-ВЫВОД

В этой главе мы продолжим рассказывать о важнейших API языка Java и рассмотрим некоторые классы пакетов `java.io` и `java.nio`. Эти пакеты содержат разнообразные средства базового ввода-вывода, а также представляют собой ту основу, на которой строятся все файловые и сетевые коммуникации в Java. Когда вы освоите азы ввода-вывода при работе с локальными файлами, мы перейдем к пакету `java.net` и рассмотрим основные концепции коммуникаций в сетях. (Самая популярная из сетей — интернет — рассматривается в главе 12.)

Мы начнем с классов для работы с потоками данных из `java.io`, которые являются subclasses базовых классов `InputStream`, `OutputStream`, `Reader` и `Writer`. Затем мы рассмотрим класс `File` и обсудим, как выполнять чтение и запись файлов с помощью классов из `java.io`. Также будут кратко рассмотрены вопросы сжатия и сериализации данных. Заодно мы представим пакет `java.nio`. Пакет NIO («New» I/O — «новый» пакет ввода-вывода) добавляет очень важную функциональность, адаптированную для создания высокопроизводительных служб, и во многих случаях предоставляет более современные, улучшенные API, которыми можно заменять некоторые из инструментов пакета `java.io`¹.

Потоки данных

Самый фундаментальный механизм ввода-вывода в Java основан на *потоках данных* (*streams*). На одном конце потока данных (по крайней мере концептуально) находится *объект записи* (*writer*), а на другом — *объект чтения* (*reader*). Работая с пакетом `java.io` для ввода и вывода на терминал, или для чтения

¹ Пакет NIO появился в Java 1.4 и его уже трудно назвать «новым», но он все-таки современнее исходного базового пакета, поэтому название прижилось.

и записи файлов, или для передачи данных через сокет, вы будете использовать разные типы потоков данных. Далее в этой главе будет рассмотрен пакет NIO, в котором есть похожая концепция *каналов* (*channels*). Одно из различий между потоками и каналами заключается в том, что потоки ориентированы на передачу байтов или символов, а каналы — на работу с «буферами», содержащими эти типы данных. Впрочем, и потоки и каналы при этом решают примерно одну и ту же задачу. Начнем с краткой классификации доступных типов потоков данных:

InputStream, OutputStream

Абстрактные классы, определяющие базовую функциональность чтения или записи неструктурированной последовательности байтов. Все остальные байтовые потоки Java создаются на основе этих двух базовых классов.

Reader, Writer

Абстрактные классы, определяющие базовую функциональность чтения или записи последовательности символьных данных, с поддержкой Unicode. Все остальные символьные потоки в Java создаются на основе Reader и Writer.

InputStreamReader, OutputStreamWriter

Классы, устанавливающие связь между байтовыми и символьными потоками данных в соответствии с заданной схемой кодирования символов. (Помните: в Unicode символ не всегда соответствует одному байту!)

DataInputStream, DataOutputStream

Специализированные фильтры потоков данных, добавляющие возможность чтения и записи многобайтовых типов данных (например, числовых примитивов и объектов String) в универсальном формате.

ObjectInputStream, ObjectOutputStream

Специализированные фильтры потоков данных, способные записывать целые группы сериализованных объектов Java и реконструировать их.

BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter

Специализированные фильтры потоков данных, добавляющие буферизацию для повышения эффективности. Буферы почти всегда используются в реальных задачах ввода-вывода.

PrintStream, PrintWriter

Специализированные потоки данных, упрощающие вывод текста.

PipedInputStream, PipedOutputStream, PipedReader, PipedWriter

«Кольцевые» потоки данных, которые могут использоваться парами для перемещения данных внутри приложения. Данные, записанные

в `PipedOutputStream` или `PipedWriter`, читаются из соответствующего объекта `PipedInputStream` или `PipedReader`.

`FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`

Реализации `InputStream`, `OutputStream`, `Reader` и `Writer`, осуществляющие чтение и запись в файлы локальной файловой системы.

Каждый поток данных в Java работает в одном направлении. Классы ввода и вывода из пакета `java.io` представляют собой конечные стороны простых потоков данных (рис. 11.1). Для двусторонней передачи данных следует использовать по одному потоку каждого типа.

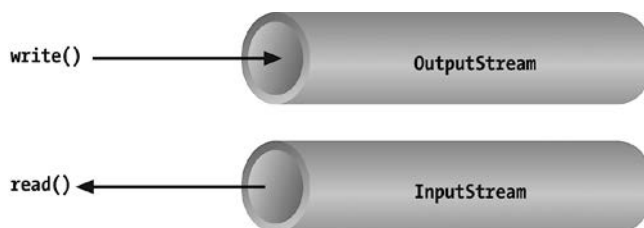


Рис. 11.1. Базовая функциональность потоков данных ввода и вывода

`InputStream` и `OutputStream` — абстрактные классы, определяющие интерфейс самого нижнего уровня для всех байтовых потоков данных. Они содержат методы для чтения и записи неструктурированного потока данных байтового уровня. Так как классы являются абстрактными, вы не сможете создать обобщенный поток ввода или вывода. Java реализует их subclasses для таких операций, как чтение и запись в файлы или взаимодействие с сокетами. Так как все байтовые потоки наследуют структуру `InputStream` или `OutputStream`, разные виды байтовых потоков могут использоваться взаимозаменяемо. Метод, у которого объявлен аргумент `InputStream`, может получить любой subclass `InputStream`. Специализированные виды потоков данных также можно накладывать или использовать в качестве оберток для базовых потоков, чтобы добавлять такую функциональность, как буферизация, фильтрация или обработка типов данных более высокого уровня.

`Reader` и `Writer` очень похожи на `InputStream` и `OutputStream`, за исключением того, что они работают с символами вместо байтов. Как полноценные символьные потоки, эти классы правильно работают с символами Unicode, что не всегда можно сказать о байтовых потоках. Часто требуется связать между собой эти символьные потоки и байтовые потоки физических устройств (например, дисков или сетей). `InputStreamReader` и `OutputStreamWriter` — специальные классы, использующие *схему кодирования символов* для преобразований между символьными и байтовыми потоками.

В этом разделе описаны все интересные типы потоков данных, кроме `FileInputStream`, `FileOutputStream`, `FileReader` и `FileWriter`. Рассмотрение файловых потоков данных мы отложили до следующего раздела, посвященного работе с файловой системой.

Базовый ввод-вывод

Классический пример объекта `InputStream` — *стандартный ввод* приложения Java. Как и поток `stdin` в С или `cin` в C++, он является источником ввода для программ командной строки (не имеющих графического интерфейса). Это входной поток той среды, в которой выполняется программа, — обычно окно терминала или вывод другой команды. Класс `java.lang.System`, обобщенный репозиторий для системных ресурсов, предоставляет ссылку на стандартный поток ввода в статической переменной `System.in`. Он также предоставляет *стандартный поток вывода* и *стандартный поток ошибок* в переменных `out` и `err` соответственно¹. Соответствие между ними показано в следующем примере:

```
InputStream stdin = System.in;
OutputStream stdout = System.out;
OutputStream stderr = System.err;
```

Этот фрагмент скрывает тот факт, что `System.out` и `System.err` являются не объектами `OutputStream`, а более специализированными и полезными объектами `PrintStream`. Мы подробнее рассмотрим их в разделе «`PrintWriter` и `PrintStream`», с. 400, а пока можем ссылаться на объекты `out` и `err` как на объекты `OutputStream`, потому что они являются производными от `OutputStream`.

Метод `read()` класса `InputStream` позволяет читать стандартный поток ввода по одному байту. Внимательно присмотревшись к API, можно заметить, что метод `read()` базового класса `InputStream` является абстрактным методом. За `System.in` лежит конкретная реализация `InputStream`, предоставляющая реальную реализацию метода `read()`:

```
try {
    int val = System.in.read();
} catch ( IOException e ) {
    ...
}
```

¹ Стандартный поток ошибок — это поток данных, обычно зарезервированный для текстовых сообщений об ошибках, которые должны выводиться для пользователя в приложениях командной строки. Он отличается от стандартного потока вывода, который часто перенаправляется в файл или передается другому приложению, оставаясь невидимым для пользователя.

Хотя мы сказали, что метод `read()` читает байтовое значение, в данном примере возвращается `int`, а не `byte`. Дело в том, что метод `read()` базовых потоков ввода в Java использует унаследованное от языка C соглашение обозначать конец потока специальным значением. Значения байтов данных возвращаются как целые числа без знака в диапазоне от 0 до 255, а специальное значение -1 означает, что был достигнут конец потока. Вы должны проверять это условие при использовании простого метода `read()`. Затем полученное значение при необходимости преобразуется в `byte`. Следующий пример читает каждый байт из потока ввода и выводит его значение:

```
try {
    int val;
    while( (val=System.in.read()) != -1 )
        System.out.println((byte)val);
} catch ( IOException e ) { ... }
```

Как видно из примеров, метод `read()` также может выдать исключение `IOException` в том случае, если при чтении из источника потока произошла ошибка. Различные subclasses `IOException` могут указывать на источник ошибки: например, это может быть файл или сетевое подключение. Кроме того, потоки более высокого уровня, которые читают типы, более сложные, чем отдельные байты, могут выдавать исключение `EOFException` («конец файла»), обозначающее неожиданный или преждевременный конец потока данных.

Перегруженная форма `read()` заполняет массив байтов максимально возможным объемом данных вплоть до емкости массива, возвращая количество прочитанных байтов:

```
byte [] buff = new byte [1024];
int got = System.in.read( buff );
```

Теоретически можно проверить количество байтов, доступных для чтения из `InputStream` в любой конкретный момент времени, при помощи метода `available()`. С этой информацией можно создать массив в точности нужного размера:

```
int waiting = System.in.available();
if ( waiting > 0 ) {
    byte [] data = new byte [ waiting ];
    System.in.read( data );
    ...
}
```

Тем не менее надежность этого способа зависит от того, насколько реализация потока способна точно определить, сколько данных может быть принято. Этот способ обычно подходит для файлов, но для других типов потоков данных на него полагаться не стоит.

Методы `read()` блокируют выполнение до того, как будут прочитаны хотя бы какие-то данные (по крайней мере один байт). В общем случае следует проверить возвращенное значение, чтобы определить, сколько данных вы получили и нужно ли продолжать чтение. (Неблокирующий ввод-вывод будет рассмотрен далее в этой главе.) Метод `skip()` класса `InputStream` дает возможность пропустить заданное количество байтов. В зависимости от реализации потока пропуск байтов может выполняться более эффективно, чем их чтение.

Метод `close()` закрывает поток и освобождает все связанные с ним системные ресурсы. По соображениям быстродействия важно, чтобы большинство типов потоков были закрыты после завершения работы с ними. В некоторых случаях потоки могут закрываться автоматически при уничтожении объектов в ходе уборки мусора, но полагаться на такое поведение не стоит. В Java 7 был добавлен синтаксис «try с ресурсами», упрощающий автоматическое закрытие потоков и других подлежащих закрытию ресурсов. Некоторые примеры такого рода приводятся в разделе «Файловые потоки данных», с. 407. Флаговый интерфейс `java.io.Closeable` помечает все типы потоков данных, каналов и сопутствующих вспомогательных классов, которые могут закрываться.

Наконец, отметим, что, помимо стандартных потоков `System.in` и `System.out`, Java предоставляет API `java.io.Console` через `System.console()`. `Console` может использоваться для чтения паролей без эхо-вывода символов на экране.

Символьные потоки данных

В ранних версиях Java некоторые разновидности `InputStream` и `OutputStream` включали методы для чтения или записи строк, но большинство из них наивно предполагали, что 16-разрядный символ Unicode эквивалентен 8-разрядному байту в потоке. Такое предположение работает только для символов Latin-1 (ISO 8859-1), но не для других кодировок, используемых в других языках. В главе 8 было показано, что класс `java.lang.String` имеет конструктор, получающий массив байтов, и соответствующий метод `getBytes()`, получающий кодировку символов в аргументе. Теоретически можно было бы воспользоваться ими для преобразования массивов байтов в символы Unicode и обратно, чтобы мы могли работать с потоками байтов, представляющими символьные данные в любом формате кодирования. К счастью, нам не придется этим заниматься: в Java есть потоки, которые делают это за нас.

Классы символьных потоков данных `Reader` и `Writer` из пакета `java.io` были представлены как потоки, работающие только с символьными данными. При работе с этими классами вы мыслите понятиями символов и строковых данных, в то время как Java занимается преобразованием байтов в конкретную кодировку символов. Как вы вскоре увидите, существуют и «прямые» реализации `Reader` и `Writer` для чтения и записи файлов на уровне байтов. Но два специализиро-

ванных класса, `InputStreamReader` и `OutputStreamWriter`, преодолевают разрыв между миром символьных потоков данных и миром байтовых потоков данных. Это, соответственно, такие реализации `Reader` и `Writer`, которые могут стать обертками для нижележащих байтовых потоков, когда надо преобразовать их в символьный поток. Схема кодирования используется для преобразования между возможными многобайтовыми закодированными значениями и символами Unicode в Java. Схема кодирования может задаваться по имени в конструкторе `InputStreamReader` или `OutputStreamWriter`. Для удобства конструктор по умолчанию применяет схему кодирования, которая используется системой по умолчанию.

Например, разберем строку из стандартного ввода и преобразуем ее в целое число. Предполагается, что байты, поступающие из `System.in`, используют системную схему кодирования по умолчанию:

```
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 `InputStreamReader` назначается оберткой для `System.in`. Этот объект чтения преобразует входящие байты `System.in` в символы по схеме кодирования по умолчанию. Затем `BufferedReader` назначается оберткой для `InputStreamReader`. `BufferedReader` добавляет метод `readLine()`, который может использоваться для получения полной строки текста (до завершителя строки — комбинации символов, специфической для конкретной платформы) в `String`. Строка затем преобразуется в целое число с использованием средств, описанных в главе 8.

Важно заметить, что мы берем байтово-ориентированный входной поток данных `System.in` и безопасно преобразуем его в `Reader` для чтения символов. Если бы мы захотели использовать кодировку, отличную от системной кодировки по умолчанию, ее можно было бы задать в конструкторе `InputStreamReader`:

```
InputStreamReader reader = new InputStreamReader( System.in, "UTF-8" );
```

Для каждого символа, прочитанного из `Reader`, объект `InputStreamReader` читает один или несколько байтов и выполняет необходимое преобразование в Unicode.

Мы вернемся к теме кодировок при обсуждении API `java.nio.charset`, который позволяет явно запрашивать и использовать кодировщики и декодировщики для буферов, содержащих символы и байты. Как `InputStreamReader`, так

и `OutputStreamWriter` может получать объект кодека `Charset`, а также имя схемы кодирования символов.

Обертки для потоков данных

А если ваши потребности не ограничиваются простым чтением и записью последовательности байтов или символов? Используйте поток данных типа «фильтр» — разновидность `InputStream`, `OutputStream`, `Reader` или `Writer`, которая служит оберткой для другого потока и дополняет его новыми возможностями. Фильтр получает целевой поток данных в аргументе конструктора и передает ему обращения после выполнения собственной дополнительной обработки. Например, можно сконструировать `BufferedInputStream` для инкапсуляции системного стандартного ввода:

```
InputStream bufferedIn = new BufferedInputStream( System.in );
```

`BufferedInputStream` — тип потока-фильтра, который осуществляет опережающее чтение и буферизует некоторый объем данных. `BufferedInputStream` накладывает дополнительный уровень функциональности на нижележащий поток. На рис.11.2 показана схема для `DataInputStream` — разновидности потока, которая может читать типы данных более высокого уровня (например, примитивы и строки Java).

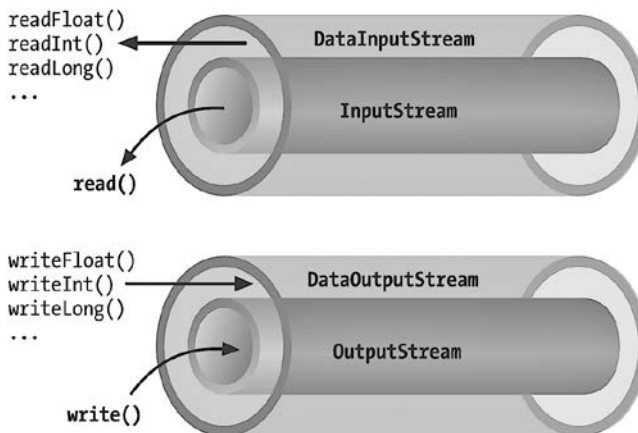


Рис. 11.2. Формирование многослойных потоков данных

Как видно из приведенного фрагмента, фильтр `BufferedInputStream` является частным случаем `InputStream`. Поскольку потоки-фильтры сами являются subclasses базовых типов потоков, они могут использоваться как аргументы

при конструировании других фильтров. Это позволяет накладывать потоки-фильтры один на другой, формируя разные комбинации функциональности. Например, можно сначала обернуть `System.in` в `BufferedInputStream`, а затем обернуть `BufferedInputStream` в `DataInputStream` для чтения специальных типов данных с буферизацией.

Следующие четыре класса позволяют создавать новые типы потоков-фильтров: `FilterInputStream`, `FilterOutputStream`, `FilterReader` и `FilterWriter`. Эти суперклассы предоставляют базовую функциональность «пустых» фильтров (фильтров, которые не делают ничего), делегируя все свои вызовы методов нижележащему потоку данных. Реальные subclasses потоков-фильтров subclassируют их и переопределяют различные методы для добавления дополнительной обработки. Далее в этой главе будет создан пример потока-фильтра.

Потоки для строк и примитивных типов

`DataInputStream` и `DataOutputStream` — потоки-фильтры, которые позволяют читать и записывать строки и примитивные типы данных, состоящие более чем из одного байта. `DataInputStream` и `DataOutputStream` реализуют интерфейсы `DataInput` и `DataOutput` соответственно. Эти интерфейсы определяют методы для чтения и записи строк и всех примитивных типов Java, включая числа и логические значения. `DataOutputStream` кодирует эти значения в машинно-независимом формате, после чего записывает их в используемый поток байтов. `DataInputStream` выполняет обратную операцию.

Вы можете сконструировать `DataInputStream` на основе `InputStream`, а затем вызвать такой метод, как `readDouble()`, для чтения примитивного типа данных:

```
DataInputStream dis = new DataInputStream( System.in );  
double d = dis.readDouble();
```

Этот код упаковывает стандартный поток ввода в обертку `DataInputStream` и использует ее для чтения значения `double`. Метод `readDouble()` читает байты из потока и конструирует `double` по прочитанным данным. Методы `DataInputStream` ожидают, что числовые типы данных передаются в *сетевом порядке байтов* (*network byte order*) — этот стандарт указывает, что старшие байты должны передаваться первыми (он также называется «обратным порядком» или «big endian», но об этом далее).

Класс `DataOutputStream` предоставляет методы записи, соответствующие методам чтения `DataInputStream`. Например, `writeInt()` записывает целое число в двоичном формате в используемый поток вывода.

Методы `readUTF()` и `writeUTF()` классов `DataInputStream` и `DataOutputStream` читают и записывают объект `String` — строку из символов Unicode, используя кодировку «формата преобразования» UTF-8. UTF-8 — очень часто исполь-

зуемая ASCII-совместимая кодировка символов Unicode. Не все кодировки гарантируют сохранение всех символов Unicode, но UTF-8 гарантирует. UTF-8 также может использоваться с потоками `Reader` и `Writer`, для чего необходимо указать имя этой кодировки.

Буферизованные потоки данных

Классы `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader` и `BufferedWriter` добавляют в поток данных буфер заданного размера. Буфер может повысить эффективность работы с потоком путем сокращения количества физических операций чтения или записи, соответствующих вызовам методов `read()` или `write()`. При создании буферизованного потока необходимо выбрать подходящий поток ввода или вывода и размер буфера. (Также возможно наложить другой поток на буферизованный поток, чтобы он выиграл от буферизации.) Пример создания буферизованного потока ввода с именем `bis`:

```
BufferedInputStream bis = new BufferedInputStream(myInputStream, 32768);
...
bis.read();
```

В этом примере задается размер буфера 32 Кбайт. Если размер буфера не указан при вызове конструктора, то автоматически выбирается разумное значение. (В настоящее время размер по умолчанию равен 8 Кбайт.) При первом вызове `read()` поток `bis` пытается заполнить данными весь 32-килобайтный буфер (если такое количество данных доступно). Последующие вызовы `read()` читают данные из буфера, который заполняется заново по мере надобности.

`BufferedOutputStream` работает аналогичным образом. Вызовы `write()` сохраняют данные в буфере; фактическая запись данных происходит только при заполнении буфера. Также можно воспользоваться методом `flush()` для того, чтобы в любой момент записать содержимое `BufferedOutputStream`. Метод `flush()` в действительности является методом самого класса `OutputStream`. Он важен, так как позволяет гарантировать, что все данные всех используемых потоков данных и потоков-фильтров были переданы (например, перед ожиданием ответа).

Некоторые входные потоки, такие как `BufferedInputStream`, позволяют пометить позицию в данных, чтобы потом к ней вернуться. Метод `mark()` устанавливает точку возврата в потоке. Он получает целочисленное значение, определяющее количество байтов, после чтения которых поток «забывает» установленную метку. Метод `reset()` возвращает поток в заданную позицию; все данные, прочитанные после вызова `mark()`, будут прочитаны заново.

Эта функциональность может пригодиться при чтении потока в парсере. Иногда при попытке разбора происходит ошибка и приходится делать другую попытку. В такой ситуации парсер может сгенерировать ошибку, после чего сбросить состояние потока в точку перед началом разбора структуры:

```
BufferedInputStream input;
...
try {
    input.mark( MAX_DATA_STRUCTURE_SIZE );
    return( parseDataStructure( input ) );
}
catch ( ParseException e ) {
    input.reset();
    ...
}
```

Классы `BufferedReader` и `BufferedWriter` работают точно так же, как и их байтовые аналоги, за исключением того, что они работают с символами вместо байтов.

PrintWriter и PrintStream

Еще один полезный поток-обертка — `java.io.PrintWriter`. Этот класс предоставляет подборку перегруженных методов `print()`, которые преобразуют свои аргументы в строки и записывают их в поток. Дополняющий набор вспомогательных методов `println()` присоединяет к строкам символ новой строки. Для вывода отформатированного текста метод `printf()` и идентичные методы `format()` позволяют записывать отформатированный текст в стиле `printf` в поток данных.

`PrintWriter` — необычный поток символов, потому что он может служить оберткой как для `OutputStream`, так и для другого `Writer`. `PrintWriter` можно назвать более мощной и функциональной альтернативой устаревшему байтовому потоку `PrintStream`. Потоки `System.out` и `System.err` являются объектами `PrintStream`; вы уже видели такие потоки в книге:

```
System.out.print("Hello, world...\n");
System.out.println("Hello, world...");
System.out.printf("The answer is %d", 17 );
System.out.println( 3.14 );
```

В ранних версиях Java не было классов `Reader` и `Writer`. Тогда использовался класс `PrintStream`, который преобразовывал байты в символы просто на основании своих предположений о кодировке символов. Во всех новых проектах следует использовать `PrintWriter`.

При создании объекта `PrintWriter` можно передать конструктору дополнительное логическое значение — флаг «автозаписи». Если это значение истинно, то `PrintWriter` автоматически вызывает `flush()` для используемого потока `OutputStream` или `Writer` при каждой отправке конца строки:

```
PrintWriter pw = new PrintWriter( myOutputStream, true /*autoFlush*/ );
pw.println("Hello!"); // Буфер потока данных автоматически записывается
                      // по концу строки
```


Когда этот способ используется с буферизованными потоками вывода, он соответствует поведению терминалов, передающих данные строку за строкой.

У `PrintStream` и `PrintWriter` есть и другое значительное преимущество перед обычными символьными потоками данных: они ограждают вас от исключений, выдаваемых используемыми потоками. В отличие от методов других классов потоков данных, методы `PrintWriter` и `PrintStream` не выдают `IOException`. Вместо этого они предоставляют метод для явной проверки ошибок в случае необходимости. Это значительно упрощает жизнь при выполнении такой пространственной операции, как вывод текста. Для проверки ошибок можно воспользоваться методом `checkError()`:

```
System.out.println( reallyLongString );
if ( System.out.checkError() ){ ... // Это ошибка
```

Класс `java.io.File`

Класс `java.io.File` инкапсулирует доступ к информации о файле или каталоге. Он нужен, чтобы получать информацию об атрибутах файла, получать список элементов каталога и выполнять базовые операции в файловой системе (например, удалить файл или создать каталог). Хотя объект `File` выполняет эти «метаоперации», он не предоставляет API для чтения и записи данных файлов; для этой цели существуют файловые потоки.

Конструкторы класса `File`

Экземпляр `File` можно создать по полному имени файла в формате `String`:

```
File fooFile = new File( "/tmp/foo.txt" );
File barDir = new File( "/tmp/bar" );
```

Также можно создать файл по относительному пути:

```
File f = new File( "foo" );
```

В этом случае Java работает относительно «текущего рабочего каталога» интерпретатора Java. Текущий рабочий каталог можно узнать из свойства `user.dir` в списке системных свойств:

```
System.getProperty( "user.dir" ); // Hanpume, "/Users/pat"
```

Перегруженная версия конструктора `File` позволяет задать путь к каталогу и имя файла в отдельных объектах `String`:

```
File fooFile = new File( "/tmp", "foo.txt" );
```

Еще одна версия позволяет задать каталог объектом `File`, а имя файла — объектом `String`:

```
File tmpDir = new File( "/tmp" ); // Объект File создается для каталога /tmp
File fooFile = new File( tmpDir, "foo.txt" );
```

Ни один из конструкторов `File` не создает файл или каталог, и создание объекта `File` для несуществующего файла не является ошибкой. Объект `File` — всего лишь дескриптор для файла или каталога, свойства которого вы хотите прочитать, записать или проверить. Например, метод экземпляра `exists()` позволяет проверить, существует ли файл или каталог.

Локализация пути

Одна из типичных проблем, возникающих при работе с файлами в Java, состоит в том, что пути должны подчиняться соглашениям локальной файловой системы. Но в файловой системе Windows в качестве корневых узлов используются буквенные обозначения дисков (C:), а разделителем для пути является обратная косая черта, или обратный слеш (\), — вместо обычной косой черты, или прямого слеша (/), в других системах.

Java пытается компенсировать эти различия. Например, на платформе Windows Java принимает пути с разделителями обоих видов: как /, так и \. Но на других платформах по-прежнему допустим только разделитель /.

Лучшее решение — позаботиться о том, чтобы вы следовали соглашениям имен локальной файловой системы. Если ваше приложение имеет графический интерфейс, который открывает и сохраняет файлы по запросу пользователя, вы сможете использовать для реализации этой функциональности класс `Swing JFileChooser`. Этот класс инкапсулирует графическое диалоговое окно выбора файлов. Методы `JFileChooser` позаботятся о системно-зависимых именах файлов за вас.

Но если ваше приложение должно работать с файлами самостоятельно, ситуация немного усложняется. Класс `File` содержит несколько статических переменных, которые помогают справиться с этой проблемой. `File.separator` определяет объект `String`, который задает разделитель путей для локального компьютера (например, / для Unix и macOS и \ для Windows); `File.separatorChar` предоставляет ту же информацию в формате `char`.

Системно-зависимую информацию можно использовать несколькими способами. Пожалуй, самый простой способ локализации путей — выбрать соглашение, которое вы будете использовать во внутренней реализации (например, /), и выполнить замену для локализованного разделителя.

```
// Прямой слеш будет нашим стандартом
String path = "mail/2004/june/merle";
path = path.replace('/', File.separatorChar);
File mailbox = new File( path );
```

Также можно работать с отдельными частями пути и формировать локальный путь только тогда, когда в нем возникнет необходимость:

```
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() );
```



Не забывайте, что Java интерпретирует литерал `\` в исходном коде как служебный символ в составе `String`. Чтобы включить обратную косую черту в `String`, необходимо использовать `\\`.

Чтобы справиться с проблемой множественных «корней» файловой системы (например, `C:\` в Windows), класс `File` предоставляет статический метод `listRoots()`, который возвращает массив объектов `File`, соответствующих корневым каталогам файловой системы. И снова в GUI-приложениях графическое диалоговое окно выбора файлов полностью избавляет вас от этой проблемы.

Операции с объектами `File`

Имеющийся объект `File` можно использовать для получения информации и для выполнения стандартных операций с файлом или каталогом, который он представляет. Существует ряд методов для запроса информации о `File`. Например, метод `isFile()` возвращает `true`, если `File` представляет обычный файл, тогда как `isDirectory()` возвращает `true`, если объект представляет каталог. Метод `isAbsolute()` сообщает, инкапсулирует ли объект `File` *абсолютный* или *относительный* путь. Абсолютный путь — системно-зависимая концепция, которая означает, что путь не зависит от рабочего каталога приложения или любого корневого устройства или диска (например, в Windows это полный путь, включающий букву диска: `C:\\Users\\pat\\foo.txt`).

Компоненты пути `File` можно получить при помощи следующих методов: `getName()`, `getPath()`, `getAbsolutePath()` и `getParent()`. Метод `getName()` возвращает объект `String` с именем файла без какой-либо информации о каталоге. Если `File` содержит абсолютный путь, то `getAbsolutePath()` возвращает этот путь. В ином случае он возвращает относительный путь, присоединенный к текущему рабочему каталогу (в попытке сформировать абсолютный путь). Метод `getParent()` возвращает родительский каталог файла или каталога.

Строка, возвращаемая `getPath()` или `getAbsolutePath()`, может не следовать соглашениям локальной файловой системы о регистре символов. Для получения собственной версии пути для конкретной файловой системы («канонической»

версии) используется метод `getCanonicalPath()`. Например, в Windows можно создать объект `File`, для которого `getAbsolutePath()` вернет `C:\Autoexec.bat`, но `getCanonicalPath()` вернет `C:\AUTOEXEC.BAT`; оба варианта в действительности указывают на один файл. Это может быть полезно, чтобы сравнить имена файлов, которые могли быть получены с разными соглашениями о регистре, или чтобы вывести имена файлов для пользователя.

Методы `lastModified()` и `setLastModified()` читают и устанавливают время модификации файла или каталога. Значение имеет тип `long` и содержит количество миллисекунд от начала эпохи (1 января 1970 г., 00:00:00 GMT). Метод `length()` возвращает размер файла в байтах.

Следующий фрагмент кода выводит информацию о файле:

```
File fooFile = new File( "/tmp/boofa" );
String type = fooFile.isFile() ? "File " : "Directory ";
String name = fooFile.getName();
long len = fooFile.length();
System.out.println( type + name + ", " + len + " bytes " );
```

Если объект `File` представляет каталог, то для получения списка файлов в каталоге можно воспользоваться методом `list()` или `listFiles()`:

```
File tmpDir = new File( "/tmp" );
String [] fileNames = tmpDir.list();
File [] files = tmpDir.listFiles();
```

Метод `list()` возвращает массив объектов `String`, содержащий имена файлов. Метод `listFiles()` возвращает массив объектов `Files`. Учтите, что ни один из этих методов не гарантирует определенного порядка (например, алфавитного). Для сортировки строк по алфавиту можно использовать API коллекций:

```
List list = Arrays.asList( fileNames );
Collections.sort(list);
```

Если `File` относится к несуществующему каталогу, то каталог создается вызовом `mkdir()` или `makedirs()`. Метод `mkdir()` создает каталог максимум на один уровень в глубину, так что все промежуточные каталоги в пути должны уже существовать. Метод `makedirs()` создает все уровни каталогов, необходимые для создания полного пути спецификации в `File`. Если каталог почему-либо не удалось создать, то метод возвращает `false`. Метод `renameTo()` используется для переименования файла или каталога, а метод `delete()` — для удаления файла или каталога.

Хотя каталог можно создать при помощи объекта `File`, это не самый распространенный способ создания файла; обычно это делается неявно, когда вы пытаетесь записать в него данные через `FileOutputStream` или `FileWriter`, как вскоре будет показано. Исключением является метод `createNewFile()`, который может ис-

пользоваться для создания нового файла нулевой длины в месте, определяемом объектом `File`. Этот метод полезен тем, что операция гарантированно будет «атомарной» в отношении других операций создания файлов в файловой системе. Метод `createNewFile()` возвращает логическое значение, которое сообщает вам, был ли файл успешно создан. Иногда оно используется как примитивное средство блокировки: «побеждает» тот, кто создает файл первым. (Пакет `NIO` поддерживает полноценные блокировки файлов, как будет показано далее.) Эта возможность может пригодиться в сочетании с методом `deleteOnExit()`, который помечает файлы, автоматически удаляемые при выходе из виртуальной машины `Java`. Эта комбинация позволяет защищать ресурсы или создавать приложения, которые могут выполняться только в одном экземпляре. Другой метод создания файлов, имеющий отношение к классу `File`, — статический метод `createTempFile()`, который создает файл в заданном месте с автоматически сгенерированным уникальным именем. Этот метод тоже эффективно работает в сочетании с `deleteOnExit()`.

Метод `toURL()` преобразует путь к файлу в объект `URL`-адреса `file:..` `URL` — абстракция, которая позволяет создать ссылку на любую разновидность объекта, находящуюся в произвольном месте интернета. Преобразование ссылки `File` в `URL` может пригодиться для согласования с более общими средствами, работающими с `URL`. Файловые `URL` также находят широкое применение в файловом `API` пакета `NIO`, где они нужны для работы с новыми типами файловых систем, напрямую реализуемых в коде `Java`.

В табл. 11.1 приведена сводка методов, предоставляемых классом `File`.

Таблица 11.1. Методы класса `File`

Метод	Возвращаемый тип	Описание
<code>canExecute()</code>	<code>boolean</code>	Файл является исполняемым?
<code>canRead()</code>	<code>boolean</code>	Файл (или каталог) доступен для чтения?
<code>canWrite()</code>	<code>boolean</code>	Файл (или каталог) доступен для записи?
<code>createNewFile()</code>	<code>boolean</code>	Создает новый файл
<code>createTempFile(String pfx, String sfx)</code>	<code>File</code>	Статический метод для создания нового файла с заданным префиксом и суффиксом в каталоге для временных файлов по умолчанию
<code>delete()</code>	<code>boolean</code>	Удаляет файл (или каталог)
<code>deleteOnExit()</code>	<code>void</code>	Файл должен быть удален исполнительной системой <code>Java</code> при завершении работы
<code>exists()</code>	<code>boolean</code>	Существует ли файл (или каталог)?

Таблица 11.1 (окончание)

Метод	Возвращаемый тип	Описание
<code>getAbsolutePath()</code>	<code>String</code>	Возвращает абсолютный путь для файла (или каталога)
<code>getCanonicalPath()</code>	<code>String</code>	Возвращает абсолютный путь для файла (или каталога), с исправленным регистром символов и преобразованными относительными элементами
<code>getFreeSpace()</code>	<code>long</code>	Возвращает размер свободного пространства (в байтах) в разделе, содержащем заданный путь, или 0 для недействительного пути
<code>getName()</code>	<code>String</code>	Возвращает имя файла (или каталога)
<code>getParent()</code>	<code>String</code>	Возвращает имя родительского каталога файла (или каталога)
<code>getPath()</code>	<code>String</code>	Возвращает путь для файла (или каталога) (не путайте с <code>toPath()</code>)
<code>getTotalSpace()</code>	<code>long</code>	Возвращает размер раздела (в байтах), содержащего заданный путь, или 0 для недействительного пути
<code>getUsableSpace()</code>	<code>long</code>	Возвращает размер доступного пользователю свободного пространства (в байтах) в разделе, содержащем заданный путь, или 0 для недействительного пути. Метод старается учитывать пользовательские разрешения записи
<code>isAbsolute()</code>	<code>boolean</code>	Является ли имя файла (или имя каталога) абсолютным?
<code>isDirectory()</code>	<code>boolean</code>	Является ли элемент каталогом?
<code>isFile()</code>	<code>boolean</code>	Является ли элемент файлом?
<code>isHidden()</code>	<code>boolean</code>	Является ли элемент скрытым? (Зависит от системы)
<code>lastModified()</code>	<code>long</code>	Возвращает время последнего изменения файла (или каталога)
<code>length()</code>	<code>long</code>	Возвращает длину файла
<code>list()</code>	<code>String[]</code>	Возвращает список файлов в каталоге
<code>listFiles()</code>	<code>File[]</code>	Возвращает содержимое каталога в виде массива объектов <code>File</code>
<code>listRoots()</code>	<code>File[]</code>	Возвращает массив корневых каталогов файловых систем, если они есть (например, <code>C:\</code> , <code>D:\</code> и т. д.)
<code>mkdir()</code>	<code>boolean</code>	Создает каталог

Метод	Возвращаемый тип	Описание
<code>mkdirs()</code>	<code>boolean</code>	Создает все каталоги в составе пути
<code>renameTo(File dest)</code>	<code>boolean</code>	Переименовывает файл (или каталог)
<code>setExecutable()</code>	<code>boolean</code>	Задаёт разрешения исполнения для файла
<code>setLastModified()</code>	<code>boolean</code>	Задаёт время последнего изменения файла (или каталога)
<code>setReadable()</code>	<code>boolean</code>	Задаёт разрешения чтения для файла
<code>setReadOnly()</code>	<code>boolean</code>	Переводит файл в состояние доступности только для чтения
<code>setWritable()</code>	<code>boolean</code>	Задаёт разрешения записи для файла
<code>toPath()</code>	<code>java.nio.file.Path</code>	Преобразует файл в путь NIO (см. описание файлового API NIO) (Не путайте с <code>getPath()</code>)
<code>toURL()</code>	<code>java.net.URL</code>	Генерирует объект URL для файла (или каталога)

Файловые потоки данных

Наверное, вам уже надоели все эти разговоры о файлах, — а ведь мы еще не записали ни одного байта! Что ж, веселье начинается. Давайте рассмотрим два фундаментальных потока данных для чтения и записи файлов: `FileInputStream` и `FileOutputStream`. Они предоставляют базовую байтово-ориентированную функциональность `InputStream` и `OutputStream`, применяемую для чтения и записи файлов. Эти потоки могут объединяться с потоками-фильтрами, описанными ранее, чтобы вы могли работать с файлами таким же образом, как и с другими средствами потоковой передачи данных.

Объект `FileInputStream` можно создать на базе имени файла в формате `String` или на базе объекта `File`:

```
FileInputStream in = new FileInputStream( "/etc/passwd" );
```

При создании `FileInputStream` исполнительная система Java пытается открыть заданный файл. Таким образом, конструкторы `FileInputStream` могут выдать исключение `FileNotFoundException`, если заданный файл не существует, или `IOException` при возникновении другой ошибки ввода-вывода. Вы должны перехватывать эти исключения в своем коде. Постарайтесь завести привычку использовать конструкцию «try с ресурсами», чтобы обеспечить автоматическое закрытие файлов после завершения работы с ними:

```
try ( FileInputStream fin = new FileInputStream( "/etc/passwd" ) ) {
    ...
    // Поток данных fin будет закрываться автоматически при выходе из блока try
}
```

Сразу после создания потока данных его метод `available()` и метод `length()` объекта `File` должны возвращать одинаковые значения.

Чтобы прочитать символы из файла через `Reader`, можно обернуть `FileInputStream` в `InputStreamReader`. Также вместо этого можно использовать класс `FileReader`, представленный для вашего удобства. `FileReader` представляет собой обычный объект `FileInputStream` с оберткой `InputStreamReader` и некоторыми настройками по умолчанию.

Представленный ниже вспомогательный класс `ListIt` отправляет содержимое файла или каталога 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)
                    System.out.println(line);
            }
            catch ( FileNotFoundException e ) {
                System.out.println( "File Disappeared" );
            }
    }
}
```

`ListIt` конструирует объект `File` по первому аргументу своей командной строки и проверяет, что объект `File` существует и доступен для чтения. Если `File` пред-

ставляет собой каталог, то `ListIt` выводит имена файлов из каталога. В ином случае `ListIt` читает и выводит файл по строкам.

Для записи файлов можно создать объект `FileOutputStream` на основе полного пути в формате `String` или объекта `File`. Однако в отличие от `FileInputStream`, конструкторы `FileOutputStream` не выдают исключение `FileNotFoundException`. Если заданный файл не существует, то `FileOutputStream` создает файл. Конструкторы `FileOutputStream` могут выдавать исключение `IOException` при возникновении других ошибок ввода-вывода, поэтому вы все равно должны обработать это исключение.

Если заданный файл существует, то `FileOutputStream` открывает его для чтения. Когда вы в дальнейшем вызовете метод `write()`, новые данные перезапишут текущее содержимое файла. Если вам надо дописать данные с конца существующего файла, используйте форму конструктора, которая получает флаг `append`:

```
FileInputStream fooOut =
    new FileOutputStream( fooFile ); // Перезапись fooFile
FileInputStream pwdOut =
    new FileOutputStream( "/etc/passwd", true ); // Флаг добавления данных
                                                // с конца файла
```

Другой способ добавления данных с конца файла основан на использовании класса `RandomAccessFile`, который мы рассмотрим чуть позже.

Как и в случае с чтением, для записи символов (вместо байтов) в файл можно обернуть `FileOutputStream` в `OutputStreamWriter`. Если вы хотите использовать схему кодирования символов по умолчанию, используйте класс `FileWriter`, который предоставляется для удобства.

Следующий пример читает строку данных из стандартного ввода и записывает ее 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 на то, как `FileWriter` упаковывается в `PrintWriter` для упрощения записи данных. Кроме того, как добропорядочные пользователи файловой системы, мы вызывали метод `close()` после завершения работы с `FileWriter`. В данном случае при закрытии `PrintWriter` также закрывается нижележащий объект `Writer`. Кроме того, здесь можно было использовать «try с ресурсами».

RandomAccessFile

Класс `java.io.RandomAccessFile` дает возможность читать и записывать данные в заданной позиции файла. `RandomAccessFile` реализует интерфейсы `DataInput` и `DataOutput`, поэтому может использоваться для чтения и записи строк и примитивных типов в заданных позициях файла так, как если бы он был `DataInputStream` и `DataOutputStream`. Но поскольку класс предоставляет произвольный, а не последовательный доступ к данным файла, он не является субклассом ни для `InputStream`, ни для `OutputStream`.

Объект `RandomAccessFile` можно создать на основе имени файла в формате `String` или на основе объекта `File`. Конструктор также получает второй аргумент `String`, который задает режим доступа к файлу. Строка `r` обозначает доступ к файлу только для чтения, а `rw` — для чтения и записи.

```
try {
    RandomAccessFile users = new RandomAccessFile( "Users", "rw" )
} catch (IOException e) { ... }
```

Когда вы создаете `RandomAccessFile` в режиме только для чтения, Java пытается открыть заданный файл. Если файл не существует, то `RandomAccessFile` выдает исключение `IOException`. Но если вы создаете `RandomAccessFile` в режиме чтения и записи, то объект создает файл в случае его отсутствия. Конструктор все еще может выдать исключение `IOException`, если произойдет другая ошибка ввода-вывода, поэтому это исключение необходимо обработать.

После того как вы создадите объект `RandomAccessFile`, вызывайте любые методы чтения или записи, как если бы вы работали с `DataInputStream` или `DataOutputStream`. При попытке записать данные в файл, доступный только для чтения, метод записи выдает исключение `IOException`.

Класс `RandomAccessFile` выделяется на фоне других классов методом `seek()`. Этот метод получает значение `long` и определяет по нему смещение в байтах для чтения и записи в файл. Метод `getFilePointer()` может использоваться для получения текущей позиции. Если вам нужно добавить данные в конец файла, определите нужную позицию методом `length()`, а затем вызовите метод `seek()` с полученным значением. Вы можете выполнять запись или позиционирование после конца файла, но чтение после конца файла невозможно. Если вы попытаетесь это сделать, то метод `read()` выдаст исключение `EOFException`.

Пример записи данных в упрощенную базу данных:

```
users.seek( userNum * RECORDSIZE );
users.writeUTF( userName );
users.writeInt( userID );
...
```

В этом примере предполагается, что строка со значением `userName` вместе со всеми последующими данными умещается в заданный размер записи базы данных.

Файловый API пакета NIO

Перейдем от первоначального, классического файлового API языка Java к новому файловому API, который появился в Java 7. Как мы уже отмечали, он может рассматриваться как замена или дополнение для классического API. Этот новый API, включенный в пакет NIO, в свое время стал частью мероприятий по улучшению быстродействия Java и переходу на более гибкий стиль ввода-вывода с поддержкой *выбираемых* и асинхронно прерываемых *каналов*. Тем не менее в контексте работы с файлами сильная сторона нового API заключается в том, что он предоставляет более полную абстракцию *файловой системы* в Java.

Кроме улучшенной поддержки существующих, реальных файловых систем, включая, например, средства копирования и перемещения файлов, управления ссылками и получения детальных атрибутов файлов (владельцы, разрешения и т. д.), новый файловый API позволяет реализовать непосредственно в Java совершенно новые виды файловых систем. Лучший пример такого рода — провайдер файловых систем ZIP, который позволяет «смонтировать» архив, то есть ZIP-файл, как файловую систему и затем работать напрямую с содержащимися в архиве файлами с помощью стандартных API — так же, как с любой другой файловой системой. Кроме того, новый файловый API предоставляет некоторые средства, которые помогают Java-разработчикам избежать множества повторов в коде. Это упрощает отслеживание изменений в дереве каталогов, обход файловой системы (паттерн «Посетитель»), подстановки в именах файлов и вспомогательные методы для чтения целых файлов в память.

Сейчас мы рассмотрим основы файлового API NIO, а в конце главы вернемся к теме буферов и каналов. В частности, речь пойдет об интерфейсах `FileChannel` и `ByteChannel`, которые вы можете рассматривать как альтернативные буферно-ориентированные потоки для чтения и записи файлов и произвольных байтовых данных.

FileSystem и Path

В пакет `java.nio.file` входят три основных элемента. Первый — это класс `FileSystem`, который представляет используемый механизм хранения информации и служит фабрикой для объектов `Path`. Второй — это класс `Path`, который представляет файл или каталог в файловой системе. Третий элемент — вспомогательный класс `Files`, который содержит большой набор статических методов

для работы с объектами `Path` и позволяет выполнять все основные операции с файлами по аналогии с классическим API.

Отправной точкой для дальнейших исследований станет класс `FileSystems` (обратите внимание на множественное число). Он является фабрикой для объектов `FileSystem`:

```
// Файловая система по умолчанию для локального компьютера
FileSystem fs = FileSystems.getDefault();

// Специальная файловая система для ZIP-файлов, без специальных свойств
Map<String,String> props = new HashMap<>();
URI zipURI = URI.create("jar:file:/Users/pat/tmp/MyArchive.zip");
FileSystem zipfs = FileSystems.newFileSystem( zipURI, props );
```

Как видно из этого фрагмента, часто мы просто приказываем файловой системе по умолчанию работать с файлами в среде управляющего компьютера, как и с классическим API. Но класс `FileSystems` также может сконструировать `FileSystem` по идентификатору URI (специальный идентификатор, похожий на URL), обозначающему тип специальной файловой системы. Здесь префикс `jar:file` обозначает протокол URI, что показывает, что мы работаем с файлом JAR или ZIP.

`FileSystem` реализует `Closeable`, и при закрытии объекта `FileSystem` также закрываются все открытые файловые каналы и другие потоковые объекты, связанные с ним. Попытка чтения или записи в эти каналы в этой точке приведет к выдаче исключения. Следует заметить, что файловую систему по умолчанию (связанную с управляющим компьютером) закрыть нельзя.

Имея объект `FileSystem`, можно использовать его как фабрику для объектов `Path`, представляющих файлы или каталоги. При конструировании объекта `Path` можно использовать строковое представление, как и с классическим объектом `File`, и в дальнейшем использовать с методами `Files` для создания, чтения, записи и удаления элементов.

```
Path fooPath = fs.getPath( "/tmp/foo.txt" );
OutputStream out = Files.newOutputStream( fooPath );
```

Этот пример открывает `OutputStream` для записи в файл `foo.txt`. По умолчанию в случае отсутствия файла он будет создан, а если файл существует, то он будет усечен (до нулевой длины) перед записью данных, однако это поведение можно изменить при помощи параметров. Методы класса `Files` будут рассмотрены в следующем разделе.

Объект `Path` реализует интерфейс `java.lang.Iterable`, который может использоваться для перебора компонентов литерала пути (например, разделенных

символом / частей tmp и foo.txt в предыдущем фрагменте). Впрочем, если вы хотите перебрать части пути для поиска других файлов или каталогов, то вас могут больше заинтересовать классы `DirectoryStream` и `FileVisitor`, которые будут рассмотрены далее. `Path` также реализует интерфейс `java.nio.file.Watchable`, позволяющий отслеживать изменения. Отслеживание изменений в ветвях файловой системы мы рассмотрим в следующем разделе.

`Path` содержит вспомогательные методы для преобразования путей относительно файла или каталога:

```
Path patPath = fs.getPath( "/User/pat/" );

Path patTmp = patPath.resolve( "tmp" ); // "/User/pat/tmp"

// То же с использованием Path
Path tmpPath = fs.getPath( "tmp" );
Path patTmp = patPath.resolve( tmpPath ); // "/User/pat/tmp"

// При преобразовании абсолютного пути относительно любого пути
// будет получен заданный путь
Path absPath = patPath.resolve( "/tmp" ); // "/tmp"

// Преобразование пути для однорангового узла (тот же родитель)
Path danPath = patPath.resolveSibling( "dan" ); // "/Users/dan"
```

Здесь показано использование методов `Path.resolve()` и `resolveSibling()` для поиска файлов или каталогов относительно заданного объекта `Path`. Метод `resolve()` обычно используется для присоединения относительного пути к существующему объекту `Path`, представляющему каталог. Если аргумент, переданный методу `resolve()`, является абсолютным путем, то результатом будет абсолютный путь (по аналогии с командой Unix или DOS `cd`). Метод `resolveSibling()` работает похожим образом, но относительно родителя целевого пути `Path`; этот метод удобен для описания цели операции `move()`.

От Path к классическим файлам и обратно

Для соединения старых и новых API в `java.io.File` и в `java.nio.file.Path` были включены методы `toPath()` и `toFile()` соответственно. Каждый из этих методов осуществляет преобразование к другой форме. Конечно, из `File` можно получить только типы `Path`, представляющие файлы и каталоги файловой системы локального компьютера по умолчанию:

```
Path tmpPath = fs.getPath( "/tmp" );
File file = tmpPath.toFile();
File tmpFile = new File( "/tmp" );
Path path = tmpFile.toPath();
```

Файловые операции NIO

С имеющимся объектом `Path` можно работать статическими методами вспомогательного класса `Files`: создать путь (файл или каталог), выполнить чтение или запись, прочитать или задать его свойства. Сначала мы перечислим эти методы, а затем обсудим самые важные из них.

В табл. 11.2 приведен список методов класса `java.nio.file.Files`. Их довольно много, поскольку класс `Files` обеспечивает все типы операций с файлами. Чтобы таблица лучше воспринималась, мы исключили перегруженные формы одного метода (с разными аргументами) и сгруппировали соответствующие и взаимосвязанные методы.

Таблица 11.2. Методы класса `Files` из пакета `NIO`

Метод	Возвращаемый тип	Описание
<code>copy()</code>	<code>long</code> или <code>Path</code>	Копирует поток данных в файловый путь, файловый путь в поток или путь в путь. Возвращает количество байтов, скопированных в целевой путь. Целевой файл может быть заменен, если он существует (по умолчанию в случае существования цели происходит ошибка). Копирование каталога приводит к созданию пустого каталога (содержимое не копируется). Копирование символической ссылки приводит к копированию данных связанного файла (обычное копирование файла)
<code>createDirectory()</code> , <code>createDirectories()</code>	<code>Path</code>	Создает один каталог или все каталоги в заданном пути. Метод <code>createDirectory()</code> выдает исключение, если каталог уже существует, тогда как метод <code>createDirectories()</code> игнорирует существующие каталоги и создает только необходимые
<code>createFile()</code>	<code>Path</code>	Создает пустой файл. Операция выполняется как атомарная и завершается успехом только в том случае, если файл не существует. (В частности, это свойство может использоваться для создания «сторожевых» файлов для защиты ресурсов и т. д.)
<code>createTempDirectory()</code> , <code>createTempFile()</code>	<code>Path</code>	Создание временного каталога или временного файла с заданным префиксом с гарантированно уникальным именем, с возможностью размещения в системном временном каталоге по умолчанию
<code>delete()</code> , <code>deleteIfExists()</code>	<code>void</code>	Удаляет файл или пустой каталог. Метод <code>deleteIfExists()</code> не выдает исключение, если файл не существует

Метод	Возвращаемый тип	Описание
<code>exists()</code> , <code>notExists()</code>	<code>boolean</code>	Проверяет, существует ли файл (<code>notExists()</code> просто возвращает обратное значение). Существует возможность указать, должен ли происходить переход по ссылкам (это делается по умолчанию)
<code>exists()</code> , <code>isDirectory()</code> , <code>isExecutable()</code> , <code>isHidden()</code> , <code>isReadable()</code> , <code>isRegularFile()</code> , <code>isWritable()</code>	<code>boolean</code>	Проверяет базовые характеристики файлов: существует ли путь, представляет ли он каталог и другие базовые атрибуты
<code>createLink()</code> , <code>createSymbolicLink()</code> , <code>isSymbolicLink()</code> , <code>readSymbolicLink()</code> , <code>createLink()</code>	<code>boolean</code> или <code>Path</code>	Создает жесткую или символическую ссылку, проверяет, является ли файл символической ссылкой, или читает целевой файл, на который указывает символическая ссылка. Символические ссылки представляют собой файлы, ссылающиеся на другие файлы. Обычные («жесткие») ссылки являются низкоуровневыми псевдонимами: два имени файла указывают на одни и те же данные. Если вы не знаете, какой тип ссылки выбрать, используйте символическую ссылку
<code>getAttribute()</code> , <code>setAttribute()</code> , <code>getFileAttributeView()</code> , <code>readAttributes()</code>	<code>Object</code> , <code>Map</code> или <code>FileAttributeView</code>	Читает или задает атрибуты файла, специфические для файловой системы: время обращения и обновления, подробные разрешения и информацию о владельце с использованием имен конкретной реализации
<code>getFileStore()</code>	<code>FileStore</code>	Получает объект <code>FileStore</code> , представляющий устройство, том или другой тип раздела файловой системы, к которому относится путь
<code>getLastModifiedTime()</code> , <code>setLastModifiedTime()</code>	<code>FileTime</code> или <code>Path</code>	Читает или задает время последнего изменения файла или каталога
<code>getOwner()</code> , <code>setOwner()</code>	<code>UserPrincipal</code>	Читает или задает объект <code>UserPrincipal</code> , представляющий владельца файла. Используйте <code>toString()</code> или <code>getName()</code> для получения строкового представления имени пользователя
<code>getPosixFilePermissions()</code> , <code>setPosixFilePermissions()</code>	<code>Set</code> или <code>Path</code>	Читает или задает полные разрешения POSIX в стиле «пользователь — группа — прочие» для пути как множество значений перечисления <code>PosixFilePermission</code>
<code>isSameFile()</code>	<code>boolean</code>	Проверяет, относятся ли два пути к одному файлу (что возможно даже в том случае, если пути не идентичны)

Таблица 11.2 (окончание)

Метод	Возвращаемый тип	Описание
<code>move()</code>	<code>Path</code>	Перемещает файл или каталог переименованием или копированием; вы можете указать, нужно ли заменять существующий файл или каталог. Используется переименование, если только копирование не требуется для перемещения файла между файловыми хранилищами или файловыми системами. Каталоги могут перемещаться этим методом только в том случае, если возможно простое переименование или если каталог пуст. Если перемещение каталога требует копирования файлов между файловыми хранилищами или файловыми системами, то метод выдает <code>IOException</code> . (В этом случае файлы необходимо копировать самостоятельно — см. <code>walkFileTree()</code>)
<code>newBufferedReader()</code> , <code>newBufferedWriter()</code>	<code>BufferedReader</code> или <code>BufferedWriter</code>	Открывает файл для чтения через <code>BufferedReader</code> или создает и открывает файл для записи через <code>BufferedWriter</code> . В обоих случаях задается кодировка символов
<code>newByteChannel()</code>	<code>SeekableByteChannel</code>	Создает новый файл или открывает существующий файл как байтовый канал с возможностью позиционирования. (См. полное обсуждение NIO далее в этой главе.) Рассмотрите альтернативную возможность использования <code>FileChannel.open()</code>
<code>newDirectoryStream()</code>	<code>DirectoryStream</code>	Возвращает <code>DirectoryStream</code> для перебора иерархии каталогов. Существует возможность передачи шаблона-«глоба» или объекта-фильтра для определения искоемых файлов
<code>newInputStream()</code> , <code>newOutputStream()</code>	<code>InputStream</code> или <code>OutputStream</code>	Открывает файл для чтения через <code>InputStream</code> или создает и открывает файл для записи через <code>OutputStream</code> . Существует возможность усечения файла для выходного потока; по умолчанию используется усечение при записи
<code>probeContentType()</code>	<code>String</code>	Возвращает MIME-тип для файла, если он может быть определен установленными службами <code>FileTypeDetector</code> , или <code>null</code> , если он неизвестен
<code>readAllBytes()</code> , <code>readAllLines()</code>	<code>byte[]</code> или <code>List<String></code>	Читает все данные из файла в <code>byte[]</code> или все символы как список строк с использованием заданной кодировки символов
<code>size()</code>	<code>long</code>	Получает размер (в байтах) файла с заданным путем

Метод	Возвращаемый тип	Описание
walkFileTree()	Path	Применяет FileVisitor к заданному дереву каталогов, с возможностью указать, нужно ли переходить по ссылкам, и задать максимальную глубину обхода
write()	Path	Записывает массив байтов или коллекцию строк (с заданной кодировкой символов) в файл с заданным путем, затем закрывает файл, с дополнительной возможностью определить поведение присоединения / усечения. По умолчанию используется усечение с записью данных

При помощи перечисленных методов можно получать входные или выходные потоки данных, а также буферизованные объекты `Reader` и `Writer` для заданного файла. Кроме того, можно создавать пути (файлы и каталоги); можно проводить перебор по иерархиям файлов. Операции с каталогами рассматриваются в следующем разделе. Напомним, что методы `resolve()` и `resolveSibling()` класса `Path` удобны для конструирования целей операций `copy()` и `move()`:

```
// Перемещение файла /tmp/foo.txt в /tmp/bar.txt
Path foo = fs.getPath( "/tmp/foo.txt" );
Files.move( foo, foo.resolveSibling("bar.txt") );
```

Для быстрого чтения и записи содержимого файлов без использования потоков данных можно воспользоваться различными методами `readAll...` и `write`, читающими и записывающими байтовые массивы или строки в файл за одну операцию. Такое решение очень удобно для файлов, легко помещающихся в память.

```
// Чтение и запись коллекции String (например, строк текста)
Charset asciiCharset = Charset.forName("US-ASCII");
List<String> csvData = Files.readAllLines( csvPath, asciiCharset );
Files.write( newCSVPath, csvData, asciiCharset );
// Чтение и запись байтов
byte [] data = Files.readAllBytes( dataPath );
Files.write( newDataPath, data );
```

Пакет NIO

В завершение нашего описания базовых средств ввода-вывода Java мы вернемся к пакету `java.nio`. Как упоминалось ранее, сокращение NIO происходит от названия «New I/O», то есть «новый ввод-вывод», а одной из задач разработки NIO было усовершенствование и расширение функциональности старого пакета `java.io`. Большая часть общей функциональности NIO действительно перекрывает ранее созданные существующие API. Но пакет NIO

разрабатывался еще и для решения конкретных проблем масштабируемости больших систем, особенно в сетевых приложениях. В следующем разделе представлены основные элементы NIO, ориентированные на работу с *буферами* и *каналами*.

Асинхронный ввод-вывод

Разработка пакета NIO была в значительной степени обусловлена необходимостью добавить в Java *неблокирующий* и *выбираемый* ввод-вывод. До появления NIO многие операции чтения и записи в Java были привязаны к потокам, поэтому приводили к блокированию потоков на непредсказуемые промежутки времени. Хотя некоторые API, например API сокетов (см. раздел «Сокеты», с. 432), предоставляют специальные средства для ограничения продолжительности вызовов ввода-вывода, это было обходным решением, компенсирующим отсутствие более общего механизма. Во многих языках, и даже в языках без прямой поддержки потоков, для повышения эффективности потоки ввода-вывода могли переводиться в неблокирующий режим, после чего проверялась их готовность к отправке или получению данных. В неблокирующем режиме операция чтения или записи выполняет только ту работу, которая может быть выполнена немедленно, — заполнение или очистку буфера, после чего происходит возврат управления. В сочетании с возможностью проверки готовности это позволяло однопоточным приложениям эффективно обслуживать сразу несколько каналов. Главный поток «выбирает» поток данных, находящийся в состоянии готовности, работает с ним, пока он не заблокируется, после чего переходит к другому. В однопроцессорной системе это практически эквивалентно использованию нескольких потоков. Оказывается, этот стиль обработки имеет преимущества по части масштабируемости даже при использовании пула потоков (вместо одного потока). Эту тему мы рассмотрим в главе 12, когда будем обсуждать веб-программирование и создание серверов, способных обслуживать множество клиентов одновременно.

Кроме неблокирующего и выбираемого ввода-вывода, пакет NIO дает возможность асинхронного закрытия и прерывания операций ввода-вывода. Как упоминалось в главе 9, до NIO не было надежных средств остановки или активизации потока, заблокированного в операции ввода-вывода. Благодаря NIO, потоки, заблокированные в операциях ввода-вывода, всегда активизируются при прерывании или при закрытии канала. Кроме того, при прерывании потока, заблокированного в операции NIO, его канал автоматически закрывается. (Закрытие канала из-за прерывания потока может не произвести впечатления, но обычно следует действовать именно так. Оставляя канал открытым, вы рискуете столкнуться с неожиданным поведением или со сторонними нежелательными манипуляциями с каналом.)

Быстродействие

Канальный ввод-вывод проектировался на основе концепции *буферов* — сложной разновидности массивов, адаптированной для сетевых коммуникаций. Пакет NIO поддерживает концепцию *прямых буферов (direct buffers)* — их область памяти находится вне виртуальной машины Java, в операционной системе управляющего компьютера. Так как все операции ввода-вывода в конечном итоге должны работать с операционной системой для обмена с памятью буфера, быстродействие некоторых операций значительно повышается. Когда требуется передавать данные между двумя внешними конечными точками, это происходит без предварительного копирования в Java и обратно.

Отображаемые и блокируемые файлы

В NIO есть два ценных инструмента общего назначения, отсутствующие в `java.io`: отображаемые в память файлы и блокировки файлов. Отображаемые в память файлы мы рассмотрим далее, а пока достаточно сказать, что с ними можно работать так, как будто эти данные находятся в памяти. Механизм блокировок файлов предусматривает возможность совместных и монопольных блокировок отдельных областей файлов, что может быть полезно при параллельных обращениях из нескольких приложений.

Каналы

Итак, `java.io` работает с потоками данных, а `java.nio` работает с каналами. *Канал* представляет собой конечную точку для передачи данных. Хотя на практике каналы похожи на потоки данных, концептуальная основа каналов более абстрактна и примитивна. Потоки данных в `java.io` определяются в понятиях ввода или вывода, с методами для чтения и записи байтов. А базовый интерфейс каналов ничего не говорит о том, как происходят коммуникации. Просто существует концепция открытия или закрытия, поддерживаемая методами `isOpen()` и `close()`. Реализации каналов для файлов, сетевых сокетов или произвольных устройств добавляют свои методы для операций чтения, записи и передачи данных. NIO предоставляет следующие каналы:

- `FileChannel`
- `Pipe.SinkChannel`, `Pipe.SourceChannel`
- `SocketChannel`, `ServerSocketChannel`, `DatagramChannel`

В этой главе будет рассматриваться `FileChannel`. Каналы `Pipe` представляют собой канальные эквиваленты функциональности `Pipe` из `java.io`. Кроме того, в Java 7 появились асинхронные версии файловых и со-

кетовых каналов: `AsynchronousFileChannel`, `AsynchronousSocketChannel`, `AsynchronousServerSocketChannel` и `AsynchronousDatagramChannel`. Эти асинхронные версии фактически буферизуют все свои операции через пул потоков и возвращают полученные результаты через асинхронный API. Асинхронный файловый канал мы рассмотрим далее в этой главе.

Все базовые каналы реализуют интерфейс `ByteChannel`, спроектированный для каналов, имеющих методы чтения и записи (как у потоков данных ввода-вывода). Однако объекты `ByteChannel` читают и записывают `ByteBuffer` вместо простых байтовых массивов.

В дополнение к этим реализациям каналов, с целью совместимости, вы можете соединять каналы с потоками ввода-вывода `java.io`, а также с объектами `Reader` и `Writer`. Но если вы будете смешивать эти возможности, то вряд ли получите все преимущества пакета NIO, в том числе повышенное быстроедействие.

Буферы

Большинство инструментов из пакетов `java.io` и `java.net` работает с байтовыми массивами. Соответствующие инструменты пакета NIO строятся на базе `ByteBuffer` (с символьным буфером `CharBuffer` для текста). Байтовые массивы очень просты, так для чего же нужны буферы? По нескольким причинам:

- **Они формализуют паттерны использования для буферизованных данных**, поддерживают такие возможности, как буферы только для чтения, отслеживают позиции чтения-записи и границы в большом буферном пространстве. Они также предоставляют средства пометки-сброса, как, например, `java.io.BufferedInputStream`.
- **Они содержат дополнительные API для работы с низкоуровневыми данными**, представляющими примитивные типы. Вы можете создавать буферы, которые рассматривают ваши байтовые данные как серию примитивов большего размера (например, `short`, `int` или `float`). Самый общий тип буфера данных, `ByteBuffer`, включает методы для чтения и записи всех примитивных типов, как и `DataOutputStream` для потоков.
- **Они абстрагируют механизм хранения данных**, позволяя среде Java применять особые оптимизации. В частности, память для прямых буферов может выделяться в виде платформенных буферов управляющей операционной системы вместо массивов в памяти Java. Каналы NIO, работающие с буферами, могут распознавать прямые буферы автоматически и пытаться оптимизировать ввод-вывод для работы с ними. Например, для чтения из файлового канала в байтовый массив Java обычно требуется копирование читаемых данных из управляющей операционной системы в память Java. С прямым буфером эти данные могут оставаться в пространстве управляю-

щей операционной системы, за пределами нормального пространства памяти Java — до того момента, когда они понадобятся.

Операции с буферами

Буфер является субклассом `java.nio.Buffer`. Базовый класс `Buffer` напоминает массив с состоянием. Он не указывает тип хранящихся в нем элементов (это должны решать подтипы), но определяет функциональность, общую для всех буферов данных. `Buffer` имеет фиксированный размер, который называется *емкостью* (*capacity*). Хотя все стандартные буферы предоставляют «произвольный доступ» к своему содержимому, `Buffer` обычно предполагает последовательное чтение и запись. Поэтому `Buffer` поддерживает концепцию текущей *позиции* (*position*), в которой выполняется чтение или запись следующего элемента. Кроме позиции, `Buffer` может поддерживать два других значения, относящихся к информации о состоянии: *лимит* (*limit*), то есть «мягкий» верхний ограничитель для чтения или записи, и *метка* (*mark*), которая может служить для запоминания более ранней позиции с целью восстановления в будущем.

Реализации `Buffer` добавляют конкретные, типизованные методы `get` и `put`, которые читают и записывают содержимое буфера. Например, `ByteBuffer` является байтовым буфером и содержит методы `get()` и `put()`, которые читают и записывают байты и массивы байтов (наряду со многими другими полезными методами, которые будут рассмотрены далее). Чтение и запись в `Buffer` изменяет маркер текущей позиции, поэтому `Buffer` отслеживает свое содержимое примерно так же, как это делает поток. Попытка чтения или записи за пределами лимита порождает исключение `BufferUnderflowException` или `BufferOverflowException` соответственно.

Значения метки, текущей позиции, лимита и емкости всегда удовлетворяют следующей формуле:

метка <= позиция <= лимит <= емкость

Позиция чтения и записи `Buffer` всегда располагается между меткой (которая служит нижней границей) и лимитом (который служит верхней границей). Емкость соответствует физическому объему памяти буфера.

Маркеры позиции и лимита могут задаваться явно при помощи методов `position()` и `limit()`. Для стандартных паттернов использования предоставляются вспомогательные методы. Метод `reset()` возвращается к позиции метки. Если метка не была установлена, выдается исключение `InvalidMarkException`. Метод `clear()` сбрасывает позицию на 0 и присваивает лимиту значение емкости, готовя буфер к новым данным (метка теряется). Обратите внимание: метод `clear()` ничего не делает с данными в буфере, он просто изменяет позиционные маркеры.

Метод `flip()` используется для общего паттерна записи данных в буфер и их последующего чтения. Он присваивает лимиту текущее значение позиции, а затем сбрасывает позицию на 0 (метка теряется), что избавляет вас от необходимости отслеживать количество прочитанных данных. Метод `rewind()` просто сбрасывает позицию на 0, оставляя лимит без изменений. Вы можете воспользоваться им для повторной записи такого же объема данных. Ниже приведен фрагмент кода, использующий эти методы для чтения данных из канала и для их записи в два канала:

```
ByteBuffer buff = ...
while ( inChannel.read( buff ) > 0 ) { // позиция = ?
    buff.flip(); // Лимит = позиция; позиция = 0
    outChannel.write( buff );
    buff.rewind(); // Позиция = 0
    outChannel2.write( buff );
    buff.clear(); // Позиция = 0; лимит = емкость
}
```

Когда вы видите этот код впервые, он кажется запутанным, потому что здесь чтение из `Channel` в действительности является записью в буфер, — и наоборот. Так как здесь записываются все имеющиеся данные вплоть до лимита, вызовы `flip()` и `rewind()` в данном случае имеют одинаковый эффект.

Типы буферов

Как упоминалось ранее, разные типы буферов добавляют методы `get` и `put` для чтения и записи отдельных типов данных. С каждым из примитивных типов Java связывается тип буфера: `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer` и `DoubleBuffer`. Каждый из них предоставляет методы для чтения и записи данных своего типа и массивов данных своего типа. Конечно, `ByteBuffer` обладает наибольшей гибкостью. Он является наиболее «разносторонним» среди всех буферов, поэтому содержит полный набор методов для чтения и записи всех остальных типов данных, а также байтов. Перечислим некоторые методы `ByteBuffer`:

```
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 упоминалось ранее, все стандартные буферы также поддерживают произвольный доступ. Для каждого из упомянутых методов `ByteBuffer` есть дополнительная форма, получающая индекс, например:

```
getLong( int индекс )
putLong( int индекс, long значение )
```

Но это еще не все. `ByteBuffer` может создать *представление* (*view*) своего содержимого в виде любого из шести других буферов класса `Buffer`. Например, метод `asShortBuffer()` возвращает объект `ShortBuffer`, являющийся «двухбайтовым» представлением исходного `ByteBuffer`. В этом случае оба объекта работают с одними и теми же данными (с одной и той же областью памяти), то есть изменения в одном из них зеркалируются в другом. Память буфера представления начинается от текущей позиции `ByteBuffer`, а его емкость определяется делением количества оставшихся байтов на размер нового типа. (Например, каждый элемент `short` занимает два байта, `float` — четыре, а `long` и `double` — восемь.) Буферы представлений удобны для чтения и записи больших блоков данных соответствующих типов внутри `ByteBuffer`.

Буферы `CharBuffer` представляют интерес в основном из-за их интеграции со `String`. Как `CharBuffer`, так и `String` реализуют интерфейс `java.lang.CharSequence`, предоставляющий стандартные методы `charAt()` и `length()`. Поэтому новые API (например, пакет `java.util.regex`) позволяют использовать `CharBuffer` вместо `String` и наоборот. В этом случае `CharBuffer` можно рассматривать как `String` с изменяемым содержимым, а также с начальной и конечной позициями, которые настраиваются программистом.

Порядок байтов

Поскольку мы говорим о чтении и записи типов, размер которых превышает байт, возникает вопрос: в каком порядке записываются байты многобайтовых значений (например, `short` и `int`)? Среди программистов есть два лагеря: сторонники «обратного порядка» («*big endian*») и «прямого порядка» («*little endian*»)¹.

¹ Названия «*big endian*» и «*little endian*» взяты из книги Дж. Свифта «Путешествия Гулливера». Так назывались два враждующих лагеря лилипутов: «тупоконечники» и «остроконечники» (в зависимости от того, с какого конца они разбивали яйца за завтраком).

Обратный порядок байтов означает, что более значащие (старшие) байты расположены сначала; при прямом порядке все наоборот. Если вы записываете двоичные данные для их последующего применения в платформенно-зависимом приложении, это может быть важно. В Intel-совместимых компьютерах принят прямой порядок, а во многих рабочих станциях с Unix — обратный порядок. Нужный выбор инкапсулируется в классе `ByteOrder`. Вы можете задать порядок байтов методом `order()` класса `ByteBuffer`, указав идентификатор `ByteOrder.BIG_ENDIAN` или `ByteOrder.LITTLE_ENDIAN`:

```
byteArray.order( ByteOrder.BIG_ENDIAN );
```

Для получения стандартного порядка байтов вашей платформы используется статический метод `ByteOrder.nativeOrder()`.

Выделение памяти для буферов

Буфер можно создать либо явным вызовом метода `allocate()`, либо инкапсуляцией существующего массива данных простого типа Java. Каждый тип буфера содержит статический метод `allocate()`, который получает емкость (размер), а также метод `wrap()`, который получает существующий массив:

```
CharBuffer cbuf = CharBuffer.allocate( 64*1024 );  
ByteBuffer bbuf = ByteBuffer.wrap( someExistingArray );
```

Прямой буфер создается аналогичным образом — методом `allocateDirect()`:

```
ByteBuffer bbuf2 = ByteBuffer.allocateDirect( 64*1024 );
```

Как упоминалось ранее, прямые буферы могут использовать структуры памяти операционной системы, оптимизированные для некоторых видов операций ввода-вывода. С другой стороны, создание прямого буфера требует чуть больше времени и ресурсов, чем создание простого буфера, поэтому эту возможность следует применять для буферов с относительно долгим жизненным циклом.

Кодировщики и декодировщики символов

Кодировщики и декодировщики преобразуют символы в байты и наоборот, выполняя отображение из стандарта Unicode в конкретные схемы кодирования. Кодировщики и декодировщики существуют в Java уже давно; они предназначены для использования потоками `Reader` и `Writer`, а также методами класса `String`, работающими с массивами байтов. Однако на первых порах не было явного API для работы с кодировками; разработчик просто ссылался на кодировщики и декодировщики по имени в формате `String`. Пакет `java.nio.charset` формализовал идею кодирования набора символов Unicode в классе `Charset`.

Класс `Charset` является фабрикой для экземпляров `Charset`, которые умеют кодировать символьные буферы в байтовые буферы и декодировать байтовые буферы в символьные. Вы можете провести поиск набора символов по имени статическим методом `Charset.forName()` и использовать его в преобразованиях.

```
Charset charset = Charset.forName("US-ASCII");
CharBuffer charBuff = charset.decode( byteBuff ); // В ASCII
ByteBuffer byteBuff = charset.encode( charBuff ); // И обратно
```

Также можно проверить доступность кодировки при помощи статического метода `Charset.isSupported()`.

Гарантированно поддерживаются следующие наборы символов:

- US-ASCII
- ISO-8859-1
- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-16

Для получения списка всех кодировщиков, доступных на вашей платформе, используется статический метод `availableCharsets()`:

```
Map map = Charset.availableCharsets();
Iterator it = map.keySet().iterator();
while ( it.hasNext() )
    System.out.println( it.next() );
```

Результат `availableCharsets()` представляет собой карту, потому что наборы символов могут иметь «синонимы» и входить в список под несколькими разными именами.

Кроме буферно-ориентированных классов из пакета `java.nio`, классы `InputStreamReader` и `OutputStreamWriter` из пакета `java.io` также были обновлены для работы с `Charset`. Кодировка задается объектом `Charset` или по имени.

CharsetEncoder и CharsetDecoder

Чтобы получить возможность более полно управлять процессом кодирования и декодирования, создайте экземпляр `CharsetEncoder` или `CharsetDecoder` (кодек) при помощи методов `newEncoder()` и `newDecoder()` класса `Charset`. В предыдущем фрагменте предполагалось, что все данные были доступны в одном буфере. Тем не менее чаще приходится обрабатывать данные, поступающие в виде блоков. Для этого API кодировщиков-декодировщиков предоставляет более общие методы

`encode()` и `decode()`, которым передается флаг — признак ожидания продолжения данных. Кодеку необходима эта информация, потому что он может остановиться на середине преобразования многобайтового символа в момент исчерпания данных. Если он знает, что скоро должны появиться новые данные, то не будет выдавать ошибку незавершенного преобразования. В следующем фрагменте декодировщик используется для чтения из буфера `bbuff` класса `ByteBuffer` и для накопления символьных данных в буфере `cbuff` класса `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();
// Использование cbuff...
```

Здесь мы отслеживаем в канале условие конца входных данных, чтобы установить флаг `done`. Обратите внимание на вызов метода `flip()` с `ByteBuffer` для ограничения объема читаемых данных и сброса позиции, обеспечивающий подготовку к операции декодирования за один шаг. Методы `encode()` и `decode()` также возвращают объект результата `CoderResult`, который может определять прогресс кодирования (в предыдущем фрагменте он не используется). Методы `isError()`, `isUnderflow()` и `isOverflow()` класса `CoderResult` сообщают, из-за чего остановилось кодирование: из-за ошибки, отсутствия байтов во входном буфере или заполнения выходного буфера соответственно.

FileChannel

Итак, мы рассмотрели основы использования каналов и буферов. Пришло время рассмотреть реальный канал. `FileChannel` — аналог `java.io.RandomAccessFile` в NIO, но он предоставляет ряд фундаментальных новых возможностей, а также повышает быстродействие. В частности, `FileChannel` предпочтительнее, чем простой файловый поток `java.io`, в тех случаях, когда вы хотите использовать блокировку файла, или доступ к отображаемому в память файлу, или хорошо оптимизированную передачу данных между файлами (а также между файлом и сетевыми каналами).

`FileChannel` можно создать на основе `Path` при помощи статического метода `open()` класса `FileChannel`:

```
FileSystem fs = FileSystems.getDefault();
Path p = fs.getPath( "/tmp/foo.txt" );
```

```
// Открыть канал по умолчанию, для чтения
try ( FileChannel channel = FileChannel.open( p ) ) {
    ...
}

// Открыть канал с параметрами, для записи
import static java.nio.file.StandardOpenOption.*;

try ( FileChannel channel = FileChannel.open( p, WRITE, APPEND, ... ) ) {
    ...
}
```

По умолчанию метод `open()` создает для файла канал, доступный только для чтения. Конечно, можно открыть канал и для записи данных (в том числе для их добавления с конца файла), а также управлять некоторыми расширенными возможностями (например, атомарным созданием и синхронизацией данных) — для этого надо передать дополнительные параметры, как показано во второй части предыдущего примера. Доступные параметры перечислены в табл. 11.3.

Таблица 11.3. Параметры `java.nio.file.StandardOpenOption`

Параметр	Описание
READ, WRITE	Открывает файл только для чтения или только для записи (по умолчанию только для чтения). Используйте оба параметра для открытия в режиме чтения-записи
APPEND	Открывает файл для записи; все записанные данные добавляются в конец файла
CREATE	Используется с WRITE для открытия файла и его создания в случае необходимости
CREATE_NEW	Используется с WRITE для атомарного создания файла; если файл уже существует, то происходит ошибка
DELETE_ON_CLOSE	Пытается удалить файл при закрытии или, если файл остается открытым, при завершении работы виртуальной машины
SYNC, DSYNC	По возможности гарантирует, что операции записи будут блокироваться, пока все данные не будут записаны на носитель. SYNC делает это для всех изменений в файлах, включая данные и метаданные (атрибуты), тогда как DSYNC добавляет это требование только для данных, содержащихся в файле
SPARSE	Используется при создании нового файла; запрашивает разреженный файл. В файловых системах, где эта возможность поддерживается, разреженные файлы имеют очень большой размер, но в основном содержат неиспользуемое пространство, которое не занимает места на носителе
TRUNCATE_EXISTING	Используется с WRITE для существующего файла; обнуляет длину файла при открытии

`FileChannel` также можно создать на основе трех традиционных классов: `FileInputStream`, `FileOutputStream` или `RandomAccessFile`:

```
FileChannel readOnlyFc = new FileInputStream("file.txt").getChannel();
FileChannel readWriteFc = new RandomAccessFile("file.txt", "rw").getChannel();
```

Каналы `FileChannel`, созданные на основе этих файловых потоков данных ввода и вывода, доступны только для чтения или только для записи соответственно. Чтобы получить `FileChannel`, доступный и для чтения, и для записи, надо использовать `RandomAccessFile` с параметрами чтения и записи как в предыдущем примере.

`FileChannel` во многом напоминает `RandomAccessFile`, но вместо байтовых массивов вы работаете с `ByteBuffer`:

```
ByteBuffer bbuf = ByteBuffer.allocate( ... );
bbuf.clear();
readOnlyFc.position( index );
readOnlyFc.read( bbuf );
bbuf.flip();
readWriteFc.write( bbuf );
```

Чтобы управлять количеством читаемых или записываемых данных, либо задайте маркеры позиции буфера и лимита, либо используйте другую форму чтения (записи), получающую начальную позицию и длину буфера. Также можно читать и записывать данные в произвольной позиции, передавая индексы методам чтения и записи:

```
readWriteFc.read( bbuf, index );
readWriteFc.write( bbuf, index2 );
```

В каждом случае фактическое число прочитанных или записанных байтов зависит от нескольких факторов. Операция пытается выполнить чтение или запись до лимита буфера, и почти всегда так и происходит при локальной работе с файлом. Операция гарантированно блокируется только после того, как будет обработан хотя бы один байт.

В любом случае возвращается количество обработанных байтов, а позиция буфера соответствующим образом обновляется, подготавливая вас к повторению операции (если процесс еще не окончен). Это одно из преимуществ работы с буферами; они могут вести отсчет за вас. Как и при работе со стандартными потоками данных, метод `read()` канала возвращает -1 при достижении конца ввода.

Размер файла всегда можно получить методом `size()`. Он может измениться при записи новых данных с конца файла. И наоборот, файл можно сократить до заданного размера методом `truncate()`.

Параллельный доступ

Каналы `FileChannel` безопасны при работе с несколькими потоками, то есть они гарантируют, что «видимые» этими потоками данные будут согласованы между каналами в одной виртуальной машине. Но если вы не задали параметры `SYNC` или `DSYNC`, у вас не будет никаких гарантий относительно того, насколько быстро записанные вашим приложением данные фактически сохранятся на диске компьютера. Если вам надо периодически проверять безопасность данных, прежде чем двигаться дальше, используйте метод `force()` для принудительной записи изменений на диск. Этот метод получает логический аргумент, указывающий, должны ли записываться метаданные файла, включая временную метку и разрешения (`SYNC` или `DSYNC`). Некоторые системы отслеживают операции чтения и записи файлов, и вы можете избежать множества обновлений, установив соответствующий флаг в состояние `false` (это означает, что немедленная синхронизация этих данных не требуется).

Как и все каналы `Channel`, `FileChannel` может быть закрыт любым потоком. После закрытия все его методы чтения-записи и методы, связанные с текущей позицией, выдают исключение `ClosedChannelException`.

Блокировка файла

`FileChannel` поддерживает монопольную и совместную блокировку отдельных областей файлов методом `lock()`:

```
FileLock fileLock = fileChannel.lock();
int start = 0, len = fileChannel2.size();
FileLock readLock = fileChannel2.lock( start, len, true );
```

Блокировки могут быть совместными или монопольными. *Монопольная* (*exclusive*) блокировка не позволяет никому другому применить любую разновидность блокировки к заданному файлу или к какой-либо области файла. *Совместная* (*shared*) блокировка позволяет другим захватывать перекрывающиеся совместные блокировки, но не монопольные блокировки. Монопольные блокировки нужны для записи, совместные — для чтения. Если вы записываете данные в файл, то любые другие операции записи в этот файл должны быть запрещены, пока ваша операция не завершится. А при чтении надо только предотвратить одновременную запись, но не одновременное чтение.

В предыдущем примере вызываемый без аргументов метод `lock()` пытается захватить монопольную блокировку для всего файла. Во второй форме этот метод получает начальную позицию и длину, а также флаг, указывающий, что блокировка в данном случае должна быть совместной (а не монопольной). Объект `FileLock`, возвращаемый методом `lock()`, может использоваться для освобождения блокировки:

```
fileLock.release();
```

Обратите внимание: блокировки файлов гарантируют только целостность данных в пределах *кооперативного API*; они не дают стопроцентной гарантии того, что помешают какой-то посторонней программе прочитать или перезаписать заблокированный файл. В общем случае гарантия соблюдения блокировки существует, только когда обе стороны пытаются захватывать блокировку и использовать ее. Кроме того, в некоторых системах не реализованы совместные блокировки; в таких случаях все запрашиваемые блокировки будут монопольными. Чтобы проверить, является ли блокировка совместной, используйте метод `isShared()`.

Блокировки `FileChannel` удерживаются до тех пор, пока канал не будет закрыт или прерван. Поэтому выполнение блокировок в команде «`try` с ресурсами» повысит надежность освобождения блокировок:

```
try ( FileChannel channel = FileChannel.open( p, WRITE ) ) {
    channel.lock();
    ...
}
```

Сетевое программирование

Сетевые коммуникации — это душа Java! Самые интересные возможности Java связаны с потенциалом динамических сетевых приложений. После многих лет совершенствования сетевых API этот язык вошел в число приоритетных для разработки традиционных клиент-серверных приложений и служб. В этом разделе мы начнем изучать пакет `java.net`, содержащий фундаментальные классы для сетевых коммуникаций и для работы с сетевыми ресурсами. Но тема сетей необъятна! В главе 12 будут рассмотрены другие средства сетевого программирования, при этом основное внимание будет уделяться функциональности, связанной с интернетом.

Классы `java.net` делятся на две основные категории. Первая — это API сокетов для работы с низкоуровневыми протоколами интернета. Вторая категория — высокоуровневые веб-ориентированные API для работы с URL-адресами (Uniform Resource Locator). На рис. 11.3 показана структура пакета `java.net`.

API сокетов Java предоставляет доступ к стандартным сетевым протоколам, используемым для передачи данных между компьютерами в интернете. Механизм сокетов лежит в основе всех остальных видов портируемых сетевых коммуникаций. Сокеты — это самый низкоуровневый инструмент во всем сетевом инструментарии. Вы можете их использовать для любых коммуникаций между клиентом и сервером, а также между одноранговыми приложениями в интернете. Но при этом вы должны реализовать ваши собственные протоколы на уровне приложений, чтобы обрабатывать и интерпретировать передаваемые данные. Высокоуровневые сетевые средства (такие, как удаленный вызов методов, HTTP и веб-службы) реализуются на основе сокетов.

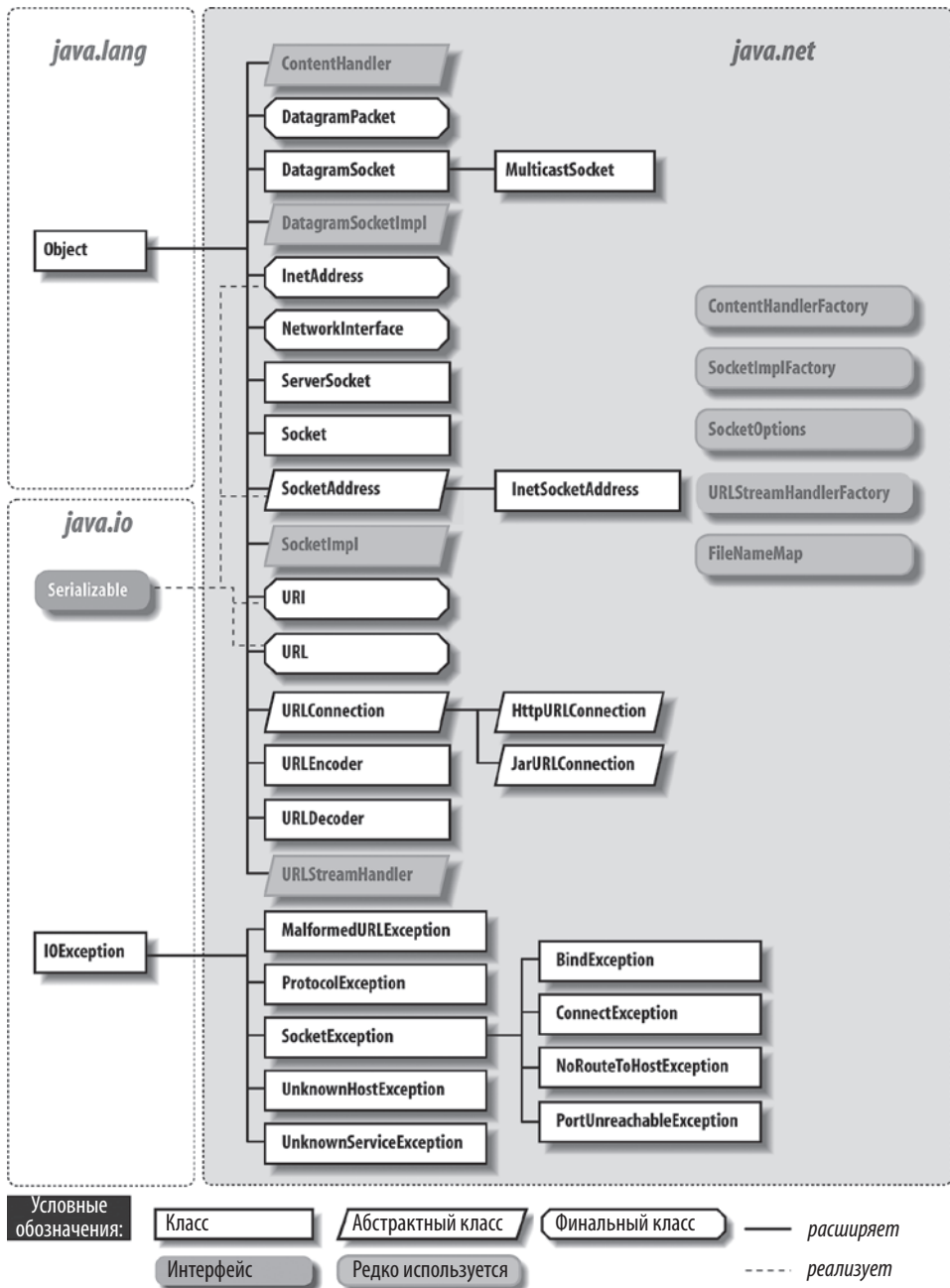


Рис. 11.3. Пакет java.net

В наши дни *веб-службами* называют более общую технологию, которая обеспечивает платформенно-независимые, слабо связанные вызовы сервисных функций на удаленных серверах с использованием таких веб-стандартов, как HTTP и JSON. Веб-службы будут рассматриваться в главе 12, посвященной веб-программированию.

В этой главе будут приведены простые практические примеры низкоуровневого и высокоуровневого сетевого программирования на основе сокетов. В главе 12 мы рассмотрим другую половину пакета `java.net`, позволяющую клиентам работать с веб-серверами и службами с помощью URL-адресов. Также в ней будут представлены сервлеты Java и инструменты, которые помогут вам написать собственные веб-приложения и службы.

Сокеты

Сокеты — это интерфейс низкоуровневого программирования сетевых коммуникаций. Сокеты передают потоки данных между приложениями, работающими на разных компьютерах (или на одном компьютере).

Сокеты появились в операционной системе BSD Unix. В некоторых языках программирования это сложные, уродливые конструкции со множеством хрупких деталей, которые так легко ломаются, что их нельзя давать в руки маленьким детям :) Причина в том, что многие API сокетов могут использоваться практически с любыми сетевыми протоколами более низкого уровня. Из-за того, что протоколы, передающие данные по сети, обладают совершенно разной функциональностью, интерфейс сокетов может быть достаточно сложным¹.

Пакет `java.net` поддерживает упрощенный объектно-ориентированный интерфейс сокетов, позволяющий легко программировать сетевые коммуникации. Если вы занимались сетевым программированием с сокетами в других языках, то вас приятно удивит, насколько простым все может быть при инкапсуляции неприятных подробностей в объектах. А если вы сталкиваетесь с сокетами впервые, то увидите, что взаимодействие с другим приложением по сети может быть таким же простым, как чтение файла или ввод данных от пользователя. Многие формы ввода-вывода в Java, включая большинство сетевых операций ввода-вывода, используют классы потоков данных (см. раздел «Потоки данных», с. 390). Потоки данных предоставляют унифицированный интерфейс ввода-вывода, чтобы чтение или запись данных по сети почти не отличались от чтения или записи в локальной системе. Кроме потоково-ориентированных

¹ Если вас заинтересует тема сокетов в целом, обратитесь к книге У. Ричарда Стивенса «Unix: разработка сетевых приложений» (СПб.: Питер).

интерфейсов, сетевые API Java могут работать с буферно-ориентированным API NIO в приложениях с высокой степенью масштабирования. В этой главе будут показаны оба варианта.

Java предоставляет сокеты для поддержки трех разных классов протоколов: `Socket`, `DatagramSocket` и `MulticastSocket`. В этом разделе мы рассмотрим простейший класс `Socket`. Он использует *надежный* протокол, *ориентированный на соединение* (*connection-oriented*). Такой протокол можно сравнить с телефонным разговором. После того как соединение установлено, два приложения могут передавать потоки данных в обоих направлениях, причем соединение сохраняется даже тогда, когда никто не ведет передачу. Так как протокол является надежным, он также гарантирует, что данные не потеряются (при необходимости они отправляются повторно) и всегда будут доставляться получателю именно в том порядке, в каком были отправлены.

Класс `DatagramSocket`, который использует *ненадежный* протокол, *не ориентированный на соединение* (*connectionless*), остается вам для самостоятельного изучения. (Можно начать с книги Эллиота Расти Хэролда (Elliott Rusty Harold) «Java Network Programming», издательство O'Reilly.) Такой протокол напоминает пересылку по почте. Приложения могут отправлять друг другу короткие сообщения, но при этом никакое сквозное соединение заранее не создается и не делаются никакие попытки сохранить исходный порядок сообщений. Не гарантируется даже сама доставка сообщений. `MulticastSocket` является разновидностью `DatagramSocket` с поддержкой *групповой рассылки* (*multicasting*) — одновременной отправки данных нескольким получателям. Работа с сокетами групповой рассылки имеет очень много общего с работой с сокетами `DatagramSocket`.

Теоретически под уровнем сокета может работать практически любой протокол (ветераны вспомнят такие названия, как Novell IPX или AppleTalk). Но сегодня на практике существует только одно актуальное семейство протоколов, используемое в интернете, и только оно поддерживается в Java — это IP (Internet Protocol). Класс `Socket` передает данные по протоколу TCP, который является дополнением к IP, ориентированным на соединение. А класс `DatagramSocket` передает данные по протоколу UDP, который не ориентирован на соединение.

Клиенты и серверы

Говоря о сетевых приложениях, часто упоминают термины «клиент» и «сервер». Различия между ними становятся все менее четкими, но обычно сторона, иницилирующая взаимодействие, считается *клиентом*, а сторона, принявшая этот запрос, считается *сервером*. В ситуации, когда два одноранговых приложения используют сокеты для взаимодействия, различия не столь важны, но для простоты мы остановимся на этом определении.

Для наших целей главное различие между клиентом и сервером заключается в том, что клиент может в любой момент создать сокет для инициации взаимодействия с серверным приложением, а сервер должен заранее подготовиться к прослушиванию входящих сообщений. Класс `java.net.Socket` представляет одну сторону подключения через сокет между клиентом и сервером. Кроме того, сервер в ожидании новых подключений от клиентов прослушивает все сообщения с помощью класса `java.net.ServerSocket`. В большинстве случаев это означает, что выступающее в качестве сервера приложение создает объект `ServerSocket` и вызывает его метод `accept()`, который блокируется в ожидании соединения. При появлении соединения метод `accept()` создает объект `Socket`, который используется сервером для взаимодействия с клиентом. Сервер может «общаться» сразу с несколькими клиентами; в этом случае по-прежнему существует только один `ServerSocket`, но сервер создает несколько объектов `Socket` — по одному для каждого клиента, как показано на рис. 11.4.

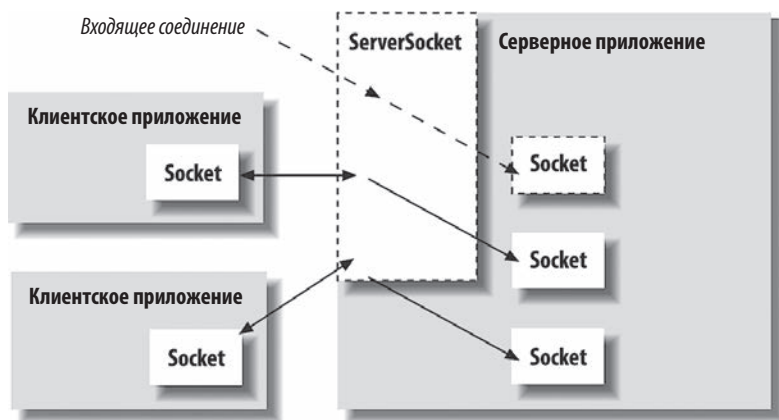


Рис. 11.4. Клиенты и серверы, Sockets и ServerSocket

На уровне сокетов клиенту нужны два элемента данных, чтобы найти в интернете сервер и соединиться с ним: *имя хоста* (*hostname*), то есть доменное имя компьютера, на котором работает серверное приложение, и *номер порта*. Имя хоста служит для поиска IP-адреса этого компьютера. Номер порта позволяет различить несколько серверов (или клиентов), работающих на одном хосте. Серверное приложение ведет прослушивание на заранее согласованном порте, ожидая запросы на соединение. Клиенты указывают номер порта, назначенный той службе, к которой они хотят подключиться. Если представить, что хосты — это гостиницы, а приложения — это постояльцы, то порты можно сравнить с номерами комнат. Чтобы один человек мог позвонить другому, он должен знать название гостиницы и номер комнаты.

Клиенты

Клиентское приложение открывает соединение с сервером, создавая объект `Socket` с указанием имени хоста и номера порта нужного сервера:

```
try {
    Socket sock = new Socket("wupost.wustl.edu", 25);
} catch ( UnknownHostException e ) {
    System.out.println("Can't find host.");
} catch ( IOException e ) {
    System.out.println("Error connecting to host.");
}
```

Этот код пытается соединиться с сокетом на 25-м порте (почтовая служба SMTP) хоста `wupost.wustl.edu`. Клиент обрабатывает возможные ситуации, когда имя хоста не удалось преобразовать в IP-адрес (`UnknownHostException`) и когда с ним не удалось установить соединение (`IOException`). В этом примере Java использует DNS (стандартный сервис преобразования имен) для преобразования имени хоста в IP-адрес. Конструктор также может получать строку с низкоуровневым IP-адресом хоста:

```
Socket sock = new Socket("22.66.89.167", 25);
```

После того как соединение будет установлено, для получения потоков данных ввода и вывода можно воспользоваться методами `getInputStream()` и `getOutputStream()` класса `Socket`. Следующий код (очень условный) отправляет и получает некие данные через потоки:

```
try {
    Socket server = new Socket("foo.bar.com", 1234);
    InputStream in = server.getInputStream();
    OutputStream out = server.getOutputStream();

    // Записать байт
    out.write(42);

    // Записать строку с завершителем (новая строка или возврат курсора)
    PrintWriter pout = new PrintWriter( out, true );
    pout.println("Hello!");

    // Прочитать байт
    byte back = (byte)in.read();

    // Прочитать строку с завершителем (новая строка или возврат курсора)
    BufferedReader bin =
        new BufferedReader( new InputStreamReader( in ) );
    String response = bin.readLine();

    server.close();
}
catch ( IOException e ) { ... }
```

В этом обмене данными клиент сначала создает объект `Socket` для взаимодействия с сервером. Конструктор `Socket` задает имя хоста сервера (`foo.bar.com`) и заранее согласованный номер порта (1234). После того как соединение будет установлено, клиент записывает один байт на сторону сервера при помощи метода `write()` класса `OutputStream`. Для упрощения отправки текстовых строк он заключает `OutputStream` в обертку `PrintWriter`. Затем выполняется ряд произвольных операций: чтение байта от сервера методом `read()` класса `InputStream` и последующее создание `BufferedReader`, из которого читается полная строка текста. Далее клиент завершает соединение методом `close()`. Все эти операции теоретически могут генерировать исключения `IOException`; наше приложение обрабатывает их в блоке `catch`.

Серверы

После установления соединения серверное приложение использует аналогичный объект `Socket` для своей стороны коммуникаций. Но чтобы принять соединение от клиента, оно должно сначала создать объект `ServerSocket`, связанный с подходящим портом. Воссоздадим предыдущий обмен данными с точки зрения сервера:

```
// Тем временем на foo.bar.com...
try {
    ServerSocket listener = new ServerSocket( 1234 );

    while ( !finished ) {
        Socket client = listener.accept(); // Ожидать соединения

        InputStream in = client.getInputStream();
        OutputStream out = client.getOutputStream();

        // Прочитать байт
        byte someByte = (byte)in.read();

        // Прочитать строку с завершителем (новая строка или возврат курсора)
        BufferedReader bin =
            new BufferedReader( new InputStreamReader( in ) );
        String someString = bin.readLine();

        // Записать байт
        out.write(43);

        // Попроцаться
        PrintWriter pout = new PrintWriter( out, true );
        pout.println("Goodbye!");

        client.close();
    }

    listener.close();
}
catch (IOException e ) { ... }
```

Сначала наш сервер создает объект `ServerSocket`, присоединенный к порту 1234. В некоторых системах установлены правила относительно того, какие порты могут использоваться приложением. Номера портов ниже 1024 обычно резервируются для системных процессов и **стандартных, хорошо известных** служб, поэтому мы выбираем любой номер за пределами этого диапазона. Объект `ServerSocket` создается только один раз; после этого можно принять любое количество входных соединений.

Затем программа входит в цикл, ожидая, пока метод `accept()` класса `ServerSocket` вернет активное соединение `Socket` от клиента. Когда соединение будет установлено, мы выполняем серверную сторону диалога, после чего закрываем соединение и возвращаемся в начало цикла для ожидания другого соединения. Наконец, когда серверное приложение хочет остановить прослушивание соединений, оно вызывает метод `close()` класса `ServerSocket`.

Этот сервер является однопоточным; он обрабатывает соединения по одному, не вызывая `accept()` для прослушивания нового подключения, пока не завершит взаимодействие с текущим соединением. Более реалистичный сервер будет содержать цикл, который принимает соединения параллельно и передает их отдельным потокам для обработки или, возможно, использует неблокирующий `ServerSocketChannel`.

Сокеты и безопасность

В предыдущих примерах предполагается, что клиент имеет разрешение для соединения с сервером, а серверу разрешено прослушивание на заданном сокете. Если вы пишете обычное автономное приложение, то эти условия обычно выполняются (и этот раздел, вероятно, можно пропустить). Однако недоверенные приложения выполняются под защитой политики безопасности, а она может устанавливать любые ограничения: с какими хостами разрешено (или запрещено) соединяться и какие порты разрешено (или запрещено) прослушивать.

Если вы собираетесь запускать свое приложение под управлением диспетчера безопасности, учитывайте, что диспетчер безопасности по умолчанию запрещает весь сетевой доступ. Таким образом, чтобы устанавливать сетевые соединения, вы должны изменить свой файл политики, чтобы предоставить соответствующие разрешения своему коду. За подробностями обращайтесь к документации Oracle (<https://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html>). Следующая запись в файле политики устанавливает разрешения для сокетов, а именно разрешает соединение с любым хостом (или от любого хоста) на любом неприлегированном порте:

```
grant {  
    permission java.net.SocketPermission  
        "*:1024-", "listen,accept,connect";  
};
```

При запуске исполнительной системы Java вы можете установить диспетчер безопасности и использовать этот файл (например, вы можете назвать его `mysecurity.policy`):

```
$ java -Djava.security.manager \
      -Djava.security.policy=mysecurity.policy MyApplication
```

Клиент DateAtHost

В прошлом на многих серверах работала простая служба времени, которая передавала локальное время своих часов на общеизвестном порте. Она была предшественником NTP, более универсального протокола сетевого времени¹. Следующий пример, `DateAtHost`, включает субкласс `java.util.Date`, который получает время с интернет-хоста по протоколу NTP, вместо того чтобы инициализировать себя показаниями локальных часов. (За информацией о классе `Date` обращайтесь к главе 8; этот класс все еще годится для некоторых целей, но в основном он уже заменен более новыми и более гибкими «родственниками» `LocalDate` и `LocalTime`.)

Наш `DateAtHost` подключается к службе времени (37-й порт) и считывает четыре байта, обозначающих время на хосте. Эти байты составлены по своеобразной спецификации, поэтому мы декодируем их, чтобы получить понятное значение времени. Пример кода:

```
// Файл: DateAtHost.java
import java.net.Socket;
import java.io.*;

public class DateAtHost extends java.util.Date {
    static int timePort = 37;
    // Кол-во секунд от начала 20 века до 00:00 GMT 1 января 1970 г.
    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 );
    }
}
```

¹ Общедоступный сайт NIST, которым мы пользуемся, настоятельно рекомендует пользователям выполнить обновление. Подробности вы можете узнать во вводной документации (<https://tf.nist.gov/tf-cgi/servers.cgi>).

Вот и все! Даже с некоторыми «излишествами» код получился не слишком длинным. Мы предоставили два возможных конструктора `DateAtHost`. Предполагается, что в обычной ситуации будет использоваться первый конструктор, который просто получает имя удаленного хоста в аргументе. Второй конструктор задает имя хоста и номер порта удаленной службы времени. (Если служба времени работает на нестандартном порте, мы используем второй конструктор для указания альтернативного номера порта.) Второй конструктор выполняет всю работу по созданию соединения и назначению времени. Первый конструктор просто вызывает второй (при помощи конструкции `this()`) с передачей порта по умолчанию в аргументе. Предоставление упрощенных конструкторов, которые вызывают своих «родственников» с аргументами по умолчанию, — распространенный и полезный паттерн в Java; это главная причина, по которой он здесь приводится.

Второй конструктор открывает сокет на заданном порте с удаленным хостом. Он создает объект `DataInputStream`, инкапсулирующий поток ввода, а затем читает четырехбайтовое целое число методом `readInt()`. При этом байты следуют в правильном порядке, и это не совпадение. Классы Java `DataInputStream` и `DataOutputStream` работают с байтами целочисленных типов в *сетевом порядке байтов* (от наиболее значащего к наименее значащему). Протокол времени (и другие стандартные сетевые протоколы, работающие с двоичными данными) также использует сетевой порядок байтов, так что никакие функции преобразования не понадобятся. Вероятно, явные преобразования данных понадобились бы при использовании нестандартного протокола, особенно при взаимодействии с клиентом или сервером, не написанным на Java. В этом случае пришлось бы читать данные байт за байтом, а затем каким-то образом переставлять их для получения четырехбайтового значения. После чтения данных работа с сокетом завершается, поэтому мы закрываем его, завершая соединение с сервером. Наконец, конструктор инициализирует оставшуюся часть объекта вызовом метода `setTime()` класса `Date` с вычисленным значением времени.

Четыре байта значения времени интерпретируются как целое число, представляющее количество секунд от начала XX века. `DateAtHost` преобразует его в формат абсолютного времени Java — количество миллисекунд, прошедших с 1 января 1970 года (произвольно выбранная дата, стандартизированная в C и Unix). Преобразование сначала создает значение `long`, которое является беззнаковым эквивалентом целочисленного времени. Оно вычитает смещение, чтобы вычислить время от начала эпохи (1 января 1970 г.) вместо начала века, и умножает значение на 1000 для перевода в миллисекунды. Преобразованное время используется для инициализации объекта.

Класс `DateAtHost` может работать с временем, полученным от удаленного хоста, почти так же легко, как `Date` работает с временем локального хоста. Добавляются только затраты на обработку возможного исключения `IOException`, которое может выдаваться конструктором `DateAtHost`:

```
try {
    Date d = new DateAtHost( "time.nist.gov" );
    System.out.println( "The time over there is: " + d );
}
catch ( IOException e ) { ... }
```

Этот пример получает с хоста `time.nist.gov` текущее время и выводит его значение.

Сетевая игра

Воспользуемся новыми навыками сетевого программирования, чтобы расширить игру с бросанием яблок для нескольких игроков. Мы сделаем это простыми средствами, и вы удивитесь, насколько быстро можно создать первый работоспособный прототип. Известно несколько механизмов соединения двух игроков для совместной игры, но мы возьмем за основу модель «клиент — сервер», которая обсуждалась в этой главе. Один пользователь запускает сервер, а другой сможет связаться с сервером, «присоединяясь» к нему в качестве клиента. Когда связь будет установлена, игроки соревнуются в том, кто быстрее справится со своими деревьями.

Подготовка пользовательского интерфейса

Начнем с добавления меню в нашу игру. Как говорилось в разделе «Меню», с. 386, все меню располагаются в строке меню и работают с объектами `ActionEvent` подобно стандартным кнопкам. Нам понадобится команда для запуска сервера и другая команда для присоединения к игре, уже запущенной кем-то другим. Основной код этих команд меню достаточно прост; мы можем воспользоваться вспомогательным методом в классе `AppleToss`:

```
private void setupNetworkMenu() {
    JMenu netMenu = new JMenu("Multiplayer");
    multiplayerHelper = new Multiplayer();

    JMenuItem startItem = new JMenuItem("Start Server");
    startItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            multiplayerHelper.startServer();
        }
    });
    netMenu.add(startItem);

    JMenuItem joinItem = new JMenuItem("Join Game...");
    joinItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String otherServer = JOptionPane.showInputDialog(AppleToss.this,
                "Enter server name or address:");
            multiplayerHelper.joinGame(otherServer);
        }
    });
}
```



```
netMenu.add(joinItem);

JMenuItem quitItem = new JMenuItem("Disconnect");
quitItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        multiplayerHelper.disconnect();
    }
});
netMenu.add(quitItem);

// Создание JMenuBar для приложения
JMenuBar mainBar = new JMenuBar();
mainBar.add(netMenu);
setJMenuBar(mainBar);
}
```

Использование анонимных внутренних классов для каждого объекта `ActionListener` из меню должно быть вам знакомо. (Или обратитесь к разделу «Ссылки на методы», с. 492, за информацией о том, как пользоваться новыми средствами Java 8 для получения более компактного кода.) Также при помощи класса `JOptionPane`, рассмотренного в разделе «Диалоговые окна ввода», с. 378, программа запрашивает у второго игрока имя или IP-адрес сервера, на котором ожидает первый игрок. Сетевая логика реализуется в отдельном классе. Класс `Mutliplayer` будет более подробно рассмотрен ниже, но вы видите набор реализуемых методов¹.

Сервер игры

Как и в разделе «Серверы», с. 436, необходимо выбрать порт и создать сокет для прослушивания входных соединений. Мы будем использовать порт 8677 (TOSS на телефонной клавиатуре). Мы можем создать внутренний класс `Server` в классе `Multiplayer`, чтобы управлять потоком, готовым к сетевым коммуникациям. Надеемся, что другие части приведенного ниже фрагмента покажутся вам знакомыми. Переменные `reader` и `writer` будут использоваться для отправки и получения игровых данных; подробнее об этом — в разделе «Протокол игры», с. 444.

```
class Server implements Runnable {
    ServerSocket listener;

    public void run() {
        Socket socket = null;
        try {
```

¹ Код игры для этой главы (находящийся в каталоге `ch11/game`) содержит метод `setupNetworkMenu()`, но слушатели в анонимных внутренних классах просто отображают диалоговое окно с информацией о выбранной команде меню. Вы можете самостоятельно создать класс `Multiplayer` и организовать вызов методов многопользовательской игры! Полный код игры, включающий сетевые части, находится в папке верхнего уровня `game` в архиве примеров книги.

```

listener = new ServerSocket(gamePort);
while (keepListening) {
    socket = listener.accept(); // Ожидать соединения

    InputStream in = socket.getInputStream();
    BufferedReader reader =
        new BufferedReader( new InputStreamReader(in) );
    OutputStream out = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(out, true);

    // ... Здесь начинается логика игрового протокола

```

Программа создает `ServerSocket`, а затем в цикле ожидает нового клиента. Хотя предполагается, что в игре может быть только один соперник, это позволяет принимать других клиентов без повторения всех действий по настройке сетевых соединений. Чтобы запустить прослушивание на сервере, достаточно создать новый поток, использующий наш класс `Server`:

```

// Из Multiplayer
Server server;

// ...

public void startServer() {
    keepListening = true;
    // ... Другое состояние игры
    server = new Server();
    serverThread = new Thread(server);
    serverThread.start();
}

```

В классе `Multiplayer` хранится ссылка на экземпляр `Server`, чтобы мы могли быстро закрыть соединение при выборе пользователем команды отключения в меню:

```

// Из Multiplayer
public void disconnect() {
    disconnecting = true;
    keepListening = false;
    // Мы находимся в середине игры и регулярно проверяем эти флаги?
    // Если нет, просто закройте сокет на сервере, чтобы прервать
    // блокирующий вызов accept()
    if (server != null && keepPlaying == false) {
        server.stopListening();
    }
    // ... Очистить другое состояние игры
}

```

Флаг `keepPlaying` в основном используется, когда мы находимся в игровом цикле, но он также будет полезен и в этом фрагменте. Если у вас имеется дей-

ствительная ссылка на сервер, но в настоящее время игра не ведется (переменная `keepPlaying` равна `false`), значит, сокет слушателя необходимо закрыть. Метод `stopListening()` во внутреннем классе `Server` достаточно прост:

```
public void stopListening() {
    if (listener != null && !listener.isClosed()) {
        try {
            listener.close();
        } catch (IOException ioe) {
            System.err.println("Error disconnecting listener: " +
                               ioe.getMessage());
        }
    }
}
```

Клиент игры

Подготовка и завершение на стороне клиента выполняются аналогично — конечно, без прослушивания `ServerSocket`. Внутренний класс `Server` заменяется внутренним классом `Client`, а для реализации клиентской логики будет создан метод `run()`:

```
class Client implements Runnable {
    String gameHost;
    boolean startNewGame;

    public Client(String host) {
        gameHost = host;
        keepPlaying = false;
        startNewGame = false;
    }

    public void run() {
        try (Socket socket = new Socket(gameHost, gamePort)) {

            InputStream in = socket.getInputStream();
            BufferedReader reader =
                new BufferedReader( new InputStreamReader( in ) );
            OutputStream out = socket.getOutputStream();
            PrintWriter writer = new PrintWriter( out, true );
            // ... Здесь начинается логика игрового протокола
        }
    }
}
```

Мы используем конструктор `Client` для передачи имени сервера, с которым будет устанавливаться соединение, и полагаемся на переменную `gamePort`, используемую `Server`, для настройки сокета прослушивания. Мы используем синтаксис «try с ресурсами», описанный в разделе «try с ресурсами», с. 221, для создания сокета и его гарантированного освобождения при завершении. В блоке `try` создаются экземпляры `reader` и `writer` для клиентской стороны диалога, как показано на рис. 11.5.

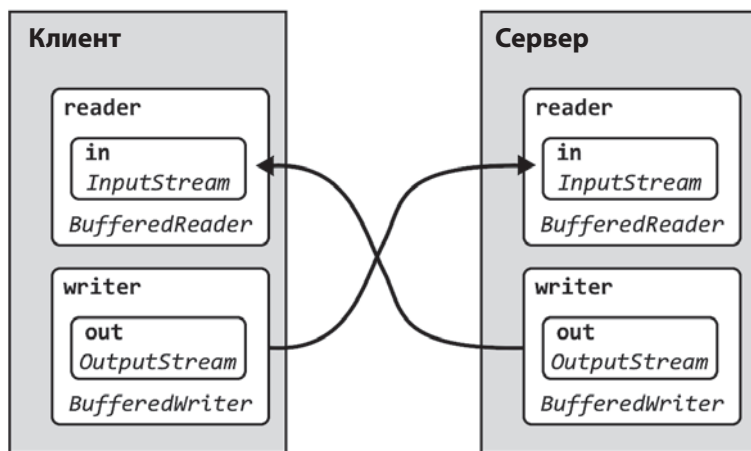


Рис. 11.5. Соединение между клиентом и сервером игры

Чтобы эта схема заработала, мы добавим во вспомогательный класс `Multiplayer` еще один удобный метод:

```
// Из Multiplayer

public void joinGame(String otherServer) {
    clientThread = new Thread(new Client(otherServer));
    clientThread.start();
}
```

В отдельном методе `disconnect()` нет необходимости, так как переменные состояния, используемые сервером, также могут управлять корректным завершением сеанса клиента. Для клиента ссылка `server` будет равна `null`, поэтому никаких попыток закрыть несуществующий слушатель не будет.

Протокол игры

Вероятно, вы заметили, что большая часть метода `run()` опущена в обоих классах, `Server` и `Client`. После создания и соединения наших потоков данных вся дальнейшая работа сводится к совместной отправке и получению информации о состояниях игры. Структурированная передача данных образует *протокол* игры. У каждой сетевой службы есть протокол (например, подумайте о значении буквы «Р» в названии HTTP). Даже наш учебный пример `DateAtHost` использует протокол (совсем простой), чтобы клиенты и серверы знали, кто должен «говорить», а кто должен «слушать» в каждый момент времени. Если обе стороны в какой-то момент будут ожидать какой-то реакции друг от друга (например, сервер и клиент заблокируются в вызове `reader.readLine()`), то соединение зависнет.

Управление ожиданиями в процессе коммуникаций занимает центральное место в любом протоколе. Но не менее важно, что следует «говорить» и как реагировать. Эта часть протокола часто требует основных усилий со стороны разработчика. Некоторые трудности связаны с тем, что во время отладки должны работать обе стороны: сервер невозможно протестировать без клиента, и наоборот. «Параллельное» программирование обеих сторон может показаться утомительным делом, но результат стоит ваших усилий. Как обычно бывает в процессе отладки, исправить ошибку при небольших пошаговых изменениях намного проще, чем искать, что могло пойти не так в большом блоке кода.

В нашем примере взаимодействием управляет сервер. Этот выбор сделан произвольно, то есть можно было возложить эту задачу на клиента или придумать более хитроумный протокол и разрешить как клиенту, так и серверу отвечать за соответствующие элементы взаимодействия. Но мы приняли решение «управляет сервер», чтобы немедленно опробовать очень простой первый шаг протокола. Сервер отправляет команду `NEW_GAME`, а затем ожидает ответа `OK` со стороны клиента. Код на стороне сервера может выглядеть примерно так:

```
// Создать новую игру с клиентом
writer.println("NEW_GAME");

// Если клиент соглашается, отправить координаты деревьев
String response = reader.readLine();
if (response != null && response.equals("OK")) {
    System.out.println("Starting a new game!")
    // ... Записать данные деревьев
} else {
    System.err.println("Unexpected start response: " + response);
    System.err.println("Skipping game and waiting again.");
    keepPlaying = false;
}
```

Получив ожидаемый ответ `OK`, можно переходить к созданию новой игры и синхронизации расположения деревьев с соперником, но об этом чуть позже. Соответствующий код на стороне клиента для первого шага имеет похожую структуру:

```
// Сначала должна поступить команда NEW_GAME
String response = reader.readLine();
// Если команда не поступила, прервать соединение и выйти
if (response == null || !response.equals("NEW_GAME")) {
    System.err.println("Unexpected initial command: " + response);
    System.err.println("Disconnecting");
    writer.println("DISCONNECT");
    return;
}
// Ура! Начинается игра. Подтвердить получение команды
writer.println("OK");
```

Если скомпилировать и запустить игру в этой точке, вы сможете запустить сервер в одной системе, а затем присоединиться к ней из второй системы. (Также можно просто запустить вторую копию игры в отдельном окне терминала. В этом случае «другим хостом» будет сетевое ключевое слово `localhost`). Почти сразу же после присоединения из второго экземпляра игры на терминале первой игры появится подтверждающее сообщение «Starting a new game!». Поздравляем! Вы только что сделали первый шаг в проектировании протокола игры. Пора двигаться дальше.

Зная, что мы начинаем новую игру, необходимо выровнять игровое поле, при этом буквально. Сервер приказывает игре сгенерировать новое поле, а затем отправляет координаты всех новых деревьев клиенту. В свою очередь, клиент получает все входящие координаты и размещает деревья на чистом поле. После того как сервер передаст всю информацию о деревьях, он отправляет команду `START` и игра начинается. Мы будем передавать сообщения в формате строк. Один из способов передачи информации о деревьях выглядит так:

```
gameField.setupNewGame();
for (Tree tree : gameField.trees) {
    writer.println("TREE " + tree.getPositionX() + " " + tree.getPositionY());
}
// ...
// Начать игру!
writer.println("START");
response = reader.readLine();
keepPlaying = response.equals("OK");
```

На стороне клиента можно в цикле вызывать `readLine()` для строк `TREE`, пока не будет получена строка `START` (с добавленной обработкой ошибок):

```
// Получить информацию о деревьях и настроить поле
gameField.trees.clear();
response = reader.readLine();
while (response.startsWith("TREE")) {
    String[] parts = response.split(" ");
    int x = Integer.parseInt(parts[1]);
    int y = Integer.parseInt(parts[2]);
    Tree tree = new Tree();
    tree.setPosition(x, y);
    gameField.trees.add(tree);
    response = reader.readLine();
}
if (!response.equals("START")) {
    // Список деревьев должен был завершиться командой START,
    // но ее не было. Прерываем
    System.err.println("Unexpected start to the game: " + response);
    System.err.println("Disconnecting");
    writer.println("DISCONNECT");
    return;
} else {
```

```
// Ура! Начинается игра. Подтвердить получение команды
writer.println("OK");
keepPlaying = true;
gameField.repaint();
}
```

В этой точке обе стороны должны иметь одинаково расположенные деревья и могут приступить к игре. Сервер входит в цикл опроса и отправляет текущий счет два раза в секунду. Клиент отвечает своим текущим счетом. Конечно, возможны и другие варианты согласования изменений в счете. Вариант с опросом прост в реализации, но в более сложных играх (или в играх, требующих немедленной обратной связи с другими игроками), скорее всего, потребуется более быстрый способ передачи данных. Но мы пока сосредоточимся на классической схеме двусторонней передачи сообщений по сети, и цикл опроса помог упростить код.

Сервер должен передавать текущий счет, пока локальный игрок не очистит все деревья или пока от клиента не будет принят запрос о завершении игры. Сервер разбирает ответ клиента, чтобы обновить его счет; при этом он отслеживает завершение игры или простой разрыв соединения. Цикл должен выглядеть примерно так:

```
while (keepPlaying) {
    try {
        if (gameField.trees.size() > 0) {
            writer.print("SCORE ");
        } else {
            writer.print("END ");
            keepPlaying = false;
        }
        writer.println(gameField.getScore(1));
        response = reader.readLine();
        if (response == null) {
            keepPlaying = false;
            disconnecting = true;
        } else {
            String parts[] = response.split(" ");
            switch (parts[0]) {
                case "END":
                    keepPlaying = false;
                case "SCORE":
                    gameField.setScore(2, parts[1]);
                    break;
                case "DISCONNECT":
                    disconnecting = true;
                    keepPlaying = false;
                    break;
                default:
                    System.err.println("Warning. Unexpected command: " +
                        parts[0] + ". Ignoring.");
            }
        }
    }
}
```

```
        Thread.sleep(500);
    } catch (InterruptedException e) {
        System.err.println("Interrupted while polling. Ignoring.");
    }
}
```

И снова клиент воспроизводит эти действия. К счастью, клиент всего лишь реагирует на команды, поступающие от сервера. Отдельный механизм опроса здесь не нужен. Клиент блокируется в ожидании чтения строки, разбирает ее, после чего формирует ответ.

```
while (keepPlaying) {
    response = reader.readLine();
    String[] parts = response.split(" ");
    switch (parts[0]) {
        case "END":
            keepPlaying = false;
        case "SCORE":
            gameField.setScore(2, parts[1]);
            break;
        case "DISCONNECT":
            disconnecting = true;
            keepPlaying = false;
            break;
        default:
            System.err.println("Unexpected game command: " +
                               response + ". Ignoring.");
    }
    if (disconnecting) {
        // Соединение разорвано одной из сторон.
        // Подтвердить и завершить
        writer.println("DISCONNECT");
        return;
    } else {
        // Если соединение не разорвано, ответить текущим счетом
        if (gameField.trees.size() > 0) {
            writer.print("SCORE ");
        } else {
            keepPlaying = false;
            writer.print("END ");
        }
        writer.println(gameField.getScore(1));
    }
}
```

Когда игрок очистит все свои деревья, он отправляет команду END, которая включает его итоговый счет. В этот момент мы спрашиваем, хотят ли игроки повторить игру. Если ответ будет положительным, можно ее продолжить с использованием тех же экземпляров `reader` и `writer` как для сервера, так и для клиента. Если нет, мы разрешаем клиенту разорвать соединение, а сервер возвращается к прослушиванию, ожидая подключения другого игрока.


```

// Если мы не разрываем соединение, предложить сыграть снова
// с тем же игроком
if (!disconnecting) {
    String message = gameField.getWinner() +
        " Would you like to ask them to play again?";
    int myPlayAgain = JOptionPane.showConfirmDialog(gameField, message,
        "Play Again?", JOptionPane.YES_NO_OPTION);

    if (myPlayAgain == JOptionPane.YES_OPTION) {
        // Если другая сторона не разрывает соединение,
        // предложить сыграть снова
        writer.println("PLAY_AGAIN");
        String playAgain = reader.readLine();
        if (playAgain != null) {
            switch (playAgain) {
                case "YES":
                    startNewGame = true;
                    break;
                case "DISCONNECT":
                    keepPlaying = false;
                    startNewGame = false;
                    disconnecting = true;
                    break;
                default:
                    System.err.println("Warning. Unexpected response: "
                        + playAgain + ". Not playing again.");
            }
        }
    }
}
}

```

И последний аналогичный фрагмент кода на стороне клиента:

```

if (!disconnecting) {
    // Проверить, хочет ли другая сторона сыграть снова
    response = reader.readLine();
    if (response != null && response.equals("PLAY_AGAIN")) {
        // А мы хотим играть снова?
        String message = gameField.getWinner() +
            " Would you like to play again?";
        int myPlayAgain = JOptionPane.showConfirmDialog(gameField, message,
            "Play Again?", JOptionPane.YES_NO_OPTION);
        if (myPlayAgain == JOptionPane.YES_OPTION) {
            writer.println("YES");
            startNewGame = true;
        } else {
            // Повторной игры не будет, разорвать соединение
            disconnecting = true;
            writer.println("DISCONNECT");
        }
    }
}
}

```

В табл. 11.4 приведен список команд нашего простого протокола.

Таблица 11.4. Протокол игры AppleToss

Команда сервера	Аргументы (необязательные)	Ответ клиента	Аргументы (необязательные)
NEW_GAME		OK	
TREE	x y		
START		OK	
SCORE	score	SCORE END DISCONNECT	score score
END	score	SCORE DISCONNECT	score
PLAY_AGAIN		YES DISCONNECT	
DISCONNECT			

Над игрой можно было бы еще серьезно потрудиться. Можно расширить протокол, чтобы он поддерживал игру с несколькими соперниками. Можно изменить цель, чтобы игрок должен был очистить деревья, а потом попасть яблоком в соперника. Можно сделать протокол более двусторонним, чтобы клиент мог инициировать некоторые обновления. Можно использовать альтернативные протоколы более низкого уровня, поддерживаемые Java (например, UDP вместо TCP). Для этого существует немало книг, посвященных программированию игр, сетевому программированию и программированию сетевых игр!

Дальнейшие исследования

Как всегда, мы предлагаем вам продолжить исследования сетей самостоятельно. Надеемся, что в этой главе вы получили представление о всесторонней поддержке сетевых коммуникаций в Java. Если вас интересует более детальное изучение каких-либо вопросов, то вы, разумеется, начнете с поиска в интернете. Тем более что сам интернет — это потрясающий пример сетевой среды. И неудивительно, что в Java есть замечательные инструменты для веб-программирования. В следующей главе мы расскажем о некоторых из них — предназначенных как для клиентских (*frontend*), так и для серверных (*backend*) приложений.

Веб-программирование

Думая об интернете, вы прежде всего представляете себе знакомые веб-сервисы и веб-приложения. Подумайте как следует, и вы вспомните о таких важных вещах, как серверы и браузеры, обеспечивающие работу этих приложений и обмен данными во всей Сети. Но главные роли в развитии интернета достались не самим этим приложениям, а тем стандартам и протоколам, на которых они основаны. Давным-давно, когда глобальная Сеть только проектировалась, в ней уже были разные способы пересылки файлов между компьютерами. Уже тогда существовали форматы документов, не уступающие HTML по функциональности. Но сначала не было универсальной модели идентификации, получения и отображения информации. Не было и универсального способа взаимодействия приложений с этой информацией по сети. Стремительный рост интернета начался благодаря формату HTML, который безраздельно властвует в качестве основы веб-документов. Многие разработчики знакомы с ним в той или иной степени. В этой главе мы поговорим о его дальнем родственнике — протоколе HTTP, который обеспечивает взаимодействие между веб-клиентами и веб-серверами. Мы продемонстрируем работу с идентификаторами URL (Uniform Resource Locators), которые являются стандартом для имен и адресов всех интернет-ресурсов. В Java есть очень простой API для работы с URL, позволяющий обращаться к нужным ресурсам в Сети. Вы узнаете, как устроены веб-клиенты — приложения, которые «общаются» с веб-серверами с помощью методов GET и POST из протокола HTTP. В разделе «Веб-приложения Java», с. 464, мы перейдем на серверную сторону и рассмотрим сервлеты и веб-службы — написанные на Java программы, которые выполняются на веб-серверах и «общаются» с клиентскими приложениями.

URL

URL (или «URL-адрес») — это адрес информационного ресурса в интернете. Он представляет собой текстовую строку, которая идентифицирует конкретный эле-

мент, сообщает, где его найти, и задает способ взаимодействия с ним или способ его загрузки из источника. URL может ссылаться на любые виды информационных ресурсов. Например, к этим ресурсам относятся статические данные (в том числе файлы в локальной файловой системе), веб-серверы, FTP-серверы и более динамические объекты вроде каналов новостей RSS или записей в базах данных. Некоторые строки URL могут идентифицировать даже такие динамические ресурсы, как открытые коммуникационные сеансы, а также адреса электронной почты.

Есть много способов обозначения конкретных элементов в Сети, а передаваемые в строках URL данные различаются в зависимости от информационных систем и протоколов, поэтому строка URL может иметь много разных форм. Самая обычная форма состоит из четырех компонентов: 1) протокол, по которому осуществляется связь с сервером; 2) имя хоста (сервера); 3) путь к элементу на хосте; 4) имя элемента.

протокол: //хост/путь/элемент

Протокол (другое название — «схема») представляет собой идентификатор вида `http` или `ftp`. *Хост* обычно соответствует доменному имени компьютера в интернете. *Путь* и *элемент* образуют уникальный полный путь, идентифицирующий объект на хосте. Кроме того, в строку URL можно включать дополнительные данные, например номера портов для протокола связи и «якорные ссылки», которые указывают на определенные разделы в пределах документа. Обычная форма строки изменяется в специализированных видах URL, таких как адреса электронной почты (`mailto`) или адреса записей в базах данных. Но в любом случае действует правило: сначала должен быть указан протокол, а за ним следует уникальный идентификатор. Некоторым из возможных URL больше подходит название URI (Uniform Resource Identifier). Строки URI определяют имя или местонахождение ресурса; строки URL составляют подмножество URI.

Почти все URL-адреса составляются по принципу иерархически организованных путей, поэтому мы иногда можем определять один URL относительно другого, который в таких случаях называется *базовым URL*. Мы используем базовый URL как отправную точку, а затем добавляем информацию, необходимую для идентификации конкретного элемента относительно этой точки. Например, базовый URL может указывать на каталог веб-сервера, а относительный URL — на имя файла в этом каталоге или в одном из его подкаталогов.

Класс URL

Перейдем к практике. В Java есть класс `URL`, который представляет URL-адреса и имеет простой API для обращения к веб-ресурсам, в том числе к документам и приложениям на серверах. Он может использовать расширяемый набор

обработчиков протоколов и контента, чтобы осуществлять обмен данными, а теоретически — еще и преобразовывать эти данные. Благодаря классу URL, вам достаточно всего нескольких строк кода, чтобы открыть соединение с сервером в сети и загрузить нужный контент. Время от времени появляются новые виды серверов и новые форматы контента, и для них в Java-сообществе могут разрабатываться дополнительные обработчики URL-адресов, позволяющие загружать и интерпретировать контент без изменения существующих приложений.

Каждый URL-адрес должен быть представлен экземпляром класса `java.net.URL`. Объект URL контролирует информацию обо всех компонентах строки URL-адреса и предоставляет методы для получения ресурса, заданного этой строкой. Мы можем указать в объекте URL полную строку с URL-адресом или составить ее из отдельных компонентов:

```
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 ) { ... }
```

Здесь два объекта URL указывают на один и тот же сетевой ресурс — документ `homepage.html` на сервере `foo.bar.com`. Существует ли этот ресурс, доступен ли он — все это станет известно лишь тогда, когда мы попытаемся к нему обратиться. Сразу после создания объект URL содержит только данные о местонахождении объекта и о том, как получить к нему доступ. Соединение с сервером пока еще не установлено. Мы можем проверить разные части URL-адреса методами `getProtocol()`, `getHost()` и `getFile()`. Мы также можем сравнить два URL методом `sameFile()` (кстати, это неудачное имя, так как URL-адрес может указывать не только на файл). Этот метод проверяет, указывают ли оба URL на один и тот же ресурс. При неумелом обращении метод `sameFile()` не дает точных ответов, зато он не только сравнивает строки URL, но и понимает, что один сервер может иметь сразу несколько имен, а также учитывает ряд других факторов. Впрочем, до загрузки ресурсов и их фактического сравнения он все же не доходит.

При создании объекта URL содержащаяся в нем строка разбирается на компоненты и из нее прежде всего выделяется компонент протокола. Если протокол указан неправильно или Java не может найти соответствующий обработчик протокола, то конструктор URL выдает исключение `MalformedURLException`. *Обработчик протокола* — это класс Java, который реализует коммуникационный протокол для обращения к URL-ресурсу. Например, если задан URL-адрес с префиксом `http`, то Java готовится использовать обработчик протокола HTTP, чтобы получить документ с веб-сервера.

Начиная с Java 7, гарантированно поддерживаются следующие обработчики протоколов URL: `http`, `https` («защищенный HTTP») и `ftp`; а также `file` (для

URL-адресов локальных файлов) и `jar` (для URL-адресов, ссылающихся на файлы внутри JAR-архивов). А за пределами этого набора возможны неожиданные проблемы, связанные с контентом и обработчиками протоколов, мы обсудим далее в этой главе.

Потоковые данные

Самый низкоуровневый и самый универсальный способ получения данных по URL-адресу — запросить из URL поток `InputStream`, вызвав метод `openStream()`. Получение данных в виде потока может быть полезно, если вам нужны постоянные обновления от динамического источника информации. С другой стороны, вам придется разбирать поток байтов самостоятельно. Работа в этом режиме в основном не отличается от работы с байтовым потоком при передаче данных через сокет, но обработчик протокола URL полностью берет на себя все взаимодействие с сервером и предоставляет вам только контентную часть транзакции. Не все виды URL совместимы с методом `openStream()`, потому что некоторые из них не ссылаются на конкретные данные; в случае несоответствия вы получите исключение `UnknownServiceException`.

Следующий код (упрощенная версия файла `Read.java` из примеров этой главы) выводит содержимое расположенного на веб-сервере HTML-файла:

```
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) { }
```

Мы запрашиваем поток `InputStream` вызовом метода `openStream()` и заключаем его в обертку `BufferedReader`, чтобы читать строки текста. Так как в URL-адресе указан протокол `http`, здесь используется обработчик протокола HTTP. Заметьте, что мы еще не говорили об обработчиках контента. Поскольку в данном случае данные читаются напрямую из входного потока, обработчик контента в этом не участвует (преобразования данных контента не происходит).

Получение контента в виде объекта

Итак, чтение необработанных данных из потока — это самый универсальный механизм загрузки контента в веб-приложениях. Метод `openStream()` оставляет

за вами разбор этих данных. Но класс URL поддерживает и другой, более сложный, модульный механизм обработки контента. Сейчас мы обсудим эту возможность, но учтите, что она не находит широкого применения из-за недостаточной стандартизации и ряда проблем, мешающих созданию новых обработчиков контента. За последние годы сообщество Java добилось некоторого успеха в стандартизации базового набора обработчиков протоколов, но не прилагало должных усилий для стандартизации обработчиков контента. Поэтому какой бы интересной ни была эта часть обсуждения, от нее мало пользы.

Если Java знает тип загружаемого контента и имеет для него соответствующий обработчик, то вы можете загрузить этот контент в виде объекта Java, вызвав из URL метод `getContent()`. В этом случае метод `getContent()` установит соединение с хостом, загрузит данные, определит их тип, а затем вызовет нужный обработчик контента, чтобы преобразовать их в объект. Java попытается определить тип контента по его MIME-типу¹, по расширению файла или даже посредством прямого анализа байтов.

Например, для условного URL `http://foo.bar.com/index.html` вызов метода `getContent()` использует обработчик протокола HTTP для загрузки данных и затем может использовать обработчик HTML-контента для их преобразования в соответствующий объект — документ. Аналогичным образом файл GIF может быть преобразован в объект AWT `ImageProducer` обработчиком GIF-контента. Если обратиться к файлу GIF по URL-адресу с протоколом FTP, то Java использует для получения данных тот же обработчик контента, но с другим обработчиком протокола.

Поскольку обработчик контента должен быть способен вернуть любой тип объекта, возвращаемым типом метода `getContent()` является `Object`. В результате всегда возникает вопрос: какой же тип объекта мы получили? Вскоре мы расскажем, как запросить у обработчика протокола MIME-тип объекта. На основании этой информации (и любой другой доступной информации о том, какой объект мы рассчитываем получить) можно преобразовать `Object` к более конкретному типу, который требуется. Например, если мы рассчитываем получить изображение, то можем преобразовать результат `getContent()` к типу `ImageProducer`:

```
try {
    ImageProducer ip = (ImageProducer)myURL.getContent();
} catch ( ClassCastException e ) { ... }
```

При попытке загрузки данных могут возникать различные ошибки. Например, `getContent()` может выдать исключение `IOException` при ошибке канала связи. Другие виды ошибок могут возникать на уровне приложения, то есть вам по-

¹ Сокращение MIME (Multipurpose Internet Mail Extensions) сложилось исторически. Возможно, термин «тип данных» был бы более подходящим.

надобится информация о том, как специфический для приложения контент и обработчики протоколов должны справляться с этими ошибками. Одной из главных проблем может стать отсутствие обработчика контента для данных того или иного конкретного MIME-типа. В таком случае метод `getContent()` вызывает специальный обработчик «неизвестного типа», который возвращает данные в виде низкоуровневого потока `InputStream` (и мы снова оказываемся в исходной точке).

В некоторых ситуациях нам может потребоваться информация об обработчике протокола. Представьте объект `URL`, который ссылается на несуществующий файл на HTTP-сервере. При запросе сервер возвращает знакомое сообщение «404 Not Found». Для таких операций, связанных с конкретным протоколом, приходится взаимодействовать с обработчиком протокола — мы займемся этим в следующем разделе.

Управление соединениями

При вызове из `URL` метода `openStream()` или `getContent()` происходит обращение к обработчику протокола и устанавливается соединение с удаленным сервером или ресурсом. Такие соединения представляются объектом `URLConnection`, подтипы которого управляют коммуникациями для разных протоколов и предоставляют дополнительные метаданные об источниках. Например, класс `HttpURLConnection` обрабатывает базовые веб-запросы, а также добавляет некоторые возможности, специфичные для HTTP (такие, как интерпретация сообщений «404 Not Found» и других ошибок веб-серверов). Класс `HttpURLConnection` будет рассмотрен далее в этой главе.

Объект `URLConnection` можно получить прямо из `URL` вызовом метода `openConnection()`. Одна из возможных операций `URLConnection` — запрос типа контента объекта перед чтением данных. Пример:

```
URLConnection connection = myURL.openConnection();
String mimeType = connection.getContentType();
InputStream in = connection.getInputStream();
```

Несмотря на свое имя, объект `URLConnection` изначально создается без установления сетевого соединения. В приведенном примере соединение будет установлено только после вызова метода `getContentType()`. Объект `URLConnection` не взаимодействует с источником, пока не будут запрошены данные или пока не будет явно вызван его метод `connect()`. Перед установлением соединения можно настраивать сетевые параметры и режимы, специфичные для протокола. Например, можно задать тайм-ауты для исходного соединения с сервером и для операций чтения:


```
URLConnection connection = myURL.openConnection();
connection.setConnectTimeout( 10000 ); // Миллисекунды
connection.setReadTimeout( 10000 ); // Миллисекунды
InputStream in = connection.getInputStream();
```

Как будет показано в разделе «Метод POST», с. 460, для получения всей информации, относящейся к специфике протокола, следует преобразовать объект `URLConnection` к его конкретному подтипу.

Проблема обработчиков

Описанная технология обработчиков контента и протоколов очень адаптивна: чтобы обрабатывать новые типы URL, достаточно добавить соответствующие классы обработчиков. Одним из интересных применений этой технологии могли бы стать браузеры, способные в реальном времени загружать из интернета любые нужные классы, чтобы обрабатывать новые специализированные виды URL. Эта идея широко рекламировалась на ранних порах развития Java. К сожалению, идею так и не воплотили на практике. В Java все еще нет API для динамической загрузки новых обработчиков контента и протоколов. Нет даже стандартного API, который мог бы определять, какие обработчики контента и протоколов существуют на заданной платформе.

В настоящее время в Java есть обработчики протоколов HTTP, HTTPS, FTP, FILE и JAR. Этот базовый набор вы, как правило, найдете в каждой версии Java, но вряд ли его можно считать достаточным, а история с обработчиками контента еще менее ясна. Например, среди стандартных классов Java нет даже обработчиков контента для HTML, GIF, PNG, JPEG и других распространенных типов данных. Более того, хотя обработчики контента и протоколов входят в Java API и являются неотъемлемой частью механизма работы с URL, конкретные обработчики контента и протоколов не определены в спецификации. И даже перечисленные обработчики протоколов, поставляемые вместе с Java, все еще входят в отдельный пакет классов Sun, а не в фундаментальный API, предназначенный для всех по умолчанию.

Итак, механизм обработчиков контента и протоколов в Java был рассчитан на дальнюю перспективу, но в итоге так и не материализовался. Идея создания браузеров, которые динамически расширяют себя для новых типов протоколов и нового контента, напоминает планы выпуска летающих автомобилей — они всегда ожидаются через несколько лет. Вам обязательно пригодится базовая технология обработчиков протоколов (особенно в нынешнем виде, после некоторой стандартизации), но для декодирования контента в ваших приложениях придется использовать другие фреймворки — более современные и специализированные.

Полезные фреймворки обработчиков

Заметьте, что идею динамической загрузки обработчиков можно осуществить и в других, похожих компонентах приложений. Например, в сообществе Java XML принято рассматривать XML как способ применения семантики (смысла) к документам, а Java — как портируемый способ обеспечить логику работы, соответствующую этой семантике. Вполне возможно написать на Java программу для просмотра XML-файлов с загружаемыми обработчиками, визуализирующими значения произвольных тегов XML.

А проблема, о которой мы рассказали, решается с помощью двух очень качественных API для работы с URL-потоками данных графики, музыки и видео. JAI (Java Advanced Imaging) содержит хорошо документированный и расширяемый набор обработчиков для большинства видов графики, а JMF (Java Media Framework) может воспроизводить основные типы аудио- и видеоконтента, встречающиеся в интернете.

Взаимодействие с веб-приложениями

Универсальными клиентами для серверных веб-приложений являются браузеры. Они загружают и отображают документы, а также предоставляют пользовательский интерфейс, основанный прежде всего на HTML, JavaScript и связанных документах. В этом разделе мы покажем, как написать на Java код для приложения-клиента, позволяющий взаимодействовать с серверными веб-приложениями через класс URL, то есть получать и отправлять данные командами GET и POST из протокола HTTP.

Обмен данными по протоколу HTTP может быть оптимальным для вашего приложения по многим причинам. Например, если вам важна совместимость с другим браузерным приложением. Или если вам нужен доступ к серверу через брандмауэр, когда прямое соединение на базе сокетов (и RMI) проблематично. HTTP — это фундамент всей глобальной сети, и несмотря на его ограничения (а скорее, из-за его простоты) он стал одним из наиболее широко поддерживаемых протоколов в мире. А использование Java на клиентской стороне (для приложений с графическим интерфейсом или без него) дает ряд преимуществ перед «чистыми» HTML-приложениями. Например, клиентское Java-приложение с графическим интерфейсом способно выполнять сложную визуализацию данных и проверку данных описанными в этой главе средствами, а также использовать разнообразные веб-сервисы по всему интернету.

Прежде всего мы рассмотрим, как отправлять на сервер данные, а конкретнее — данные из форм ввода на HTML-страницах. Данные из заполненных форм в парах «имя — значение» кодируются браузером в специальном формате и отправляются

на сервер одним из двух методов. Первый метод, использующий команду протокола HTTP GET, встраивает эти данные в строку URL и запрашивает этой строкой соответствующий документ. Сервер понимает, что первая часть URL относится к определенной программе, вызывает ее и передает ей в качестве параметра информацию, закодированную в URL. Второй метод использует команду протокола HTTP POST, которая просит сервер принять закодированные данные и передать их программе в виде потока данных. В Java-коде можно создать URL, который ссылается на определенную программу на стороне сервера и отправляет ей данные методами GET или POST. В разделе «Веб-приложения Java», с. 464, будет показано, как писать веб-приложения, реализующие другую сторону этого взаимодействия.

Метод GET

Метод GET для кодирования данных в URL-адрес достаточно прост. Все, что вам нужно, — создать URL, ссылающийся на программу на сервере, и воспользоваться простой схемой для встраивания в этот URL закодированных пар «имя — значение», из которых эти данные состоят. Например, следующий фрагмент кода создает URL для традиционной CGI-программы с именем `login.cgi`, расположенной на сервере `myhost`, и передает ей две пары «имя — значение». Затем выводится текст, возвращенный CGI-программой:

```
URL url = new URL(  
    // Эта строка должна быть закодирована в URL  
    "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 сформировать URL с параметрами, мы начинаем с базового URL, указывающего на `login.cgi`. К нему добавляется вопросительный знак (?), обозначающий начало параметров-данных. Далее следует первая пара «имя — значение». В URL-строку можно включить сколько угодно таких пар, разделенных символом &. Остальная часть кода просто открывает поток данных и читает ответ от сервера. Помните, что при создании URL соединение еще не устанавливается. В данном случае соединение создается неявно при вызове метода `openStream()`. Хотя предполагается, что в данном случае сервер возвращает текст, на самом деле он может отправлять любую информацию.

Важно заметить, что здесь был пропущен один шаг. Этот код работает, потому что пары «имя — значение» состоят из простого текста. Но если в парах встречаются

любые «непечатаемые» или специальные символы (в том числе ? или &), то их надо сначала закодировать. Класс `java.net.URLEncoder` предоставляет средства для кодирования таких данных. Работу с этим классом мы покажем в следующем примере кода, в разделе «Метод POST», с. 460.

Есть и другой важный момент: в этом маленьком примере показана отправка логина и пароля, но в реальных приложениях вы никогда не должны передавать конфиденциальные данные методом GET — он слишком примитивен для этого. Во-первых, он передает данные в интернет в виде простого текста, без шифрования. Во-вторых, все переданные таким образом данные сохраняются повсюду, где записываются пользовательские URL-запросы, в том числе в журналах (логах) сервера, в историях браузеров и в закладках браузеров. О безопасных веб-коммуникациях мы расскажем далее, когда перейдем к написанию веб-приложений с сервлетами.

Метод POST

Для передачи на сервер конфиденциального контента или больших объемов данных в протоколе HTTP есть метод POST. Ниже приведен код маленького приложения, которое работает как форма ввода HTML. Оно берет данные из двух текстовых полей (для имени и пароля) и отправляет их по заданному URL-адресу методом POST. Это основанное на Swing клиентское приложение «общается» с серверным веб-приложением именно так, как это сделал бы браузер:

```
// Файл: ch12/Post.java
package ch12;

import java.net.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Маленькое графическое приложение, демонстрирующее
 * метод POST протокола HTTP. Введите в командной строке URL-адрес хоста,
 * поддерживающего POST, и нажмите кнопку "Post", чтобы отправить
 * имя и пароль по этому URL-адресу
 *
 * Далее в этой главе приведен пример
 * сервлета ShowParameters, играющего роль серверной стороны соединения
 */
public class Post extends JPanel implements ActionListener {
    JTextField nameField;
    JPasswordField 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("Name ", JLabel.TRAILING), 0, 0 );
    addGB( nameField = new JTextField(20), 1, 0 );
    addGB( new JLabel("Password ", 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( ) {
    StringBuilder sb = new StringBuilder();
    String pw = new String(passwordField.getPassword());
    try {
        sb.append( URLEncoder.encode("Name", "UTF-8") + "=" );
        sb.append( URLEncoder.encode(nameField.getText(), "UTF-8") );
        sb.append( "&" + URLEncoder.encode("Password", "UTF-8") + "=" );
        sb.append( URLEncoder.encode(pw, "UTF-8") );
    } catch (UnsupportedEncodingException uee) {
        System.out.println(uee);
    }
    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( );
    // Отправка завершилась успешно?
    if ( urlcon.getResponseCode() == HttpURLConnection.HTTP_OK )
        System.out.println("Posted ok!");
    else {
        System.out.println("Bad post...");
        return;
    }
    // Ура! Сделать следующий шаг и прочитать результаты...
    // InputStream in = urlcon.getInputStream( );
    // ...

} catch (MalformedURLException e) {
    System.out.println(e); // Плохой postURL
} catch (IOException e2) {
    System.out.println(e2); // Ошибка ввода-вывода
}
}

public static void main( String [] args ) {
    if (args.length != 1) {
        System.err.println("Must specify URL on command line. Exiting.");
        System.exit(1);
    }
    JFrame frame = new JFrame("SimplePost");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.add( new Post(args[0]), "Center" );
    frame.pack();
    frame.setVisible(true);
}
}

```

При запуске этого приложения надо указать URL-адрес серверной программы в командной строке. Пример:

```
$ java Post http://www.myserver.example/cgi-bin/login.cgi
```

В начале приложения создается форма с элементами Swing, как это делалось в главе 10. Самое интересное происходит в защищенном методе `postData()`. Сначала мы создаем объект `StringBuilder` (несинхронизированная версия `StringBuffer`) и заполняем его парами «имя — значение», которые разделены символами `&`. (Начальный вопросительный знак для метода `POST` не нужен, потому что данные не встраиваются в строку URL.) Каждая пара предварительно кодируется при помощи статического метода `URLEncoder.encode()`. Мы пропускаем через этот кодировщик как поля имен, так и поля значений, несмотря на то что они не содержат специальных символов.

Затем устанавливается соединение с серверной программой. В предыдущем примере для отправки данных ничего особенного делать не приходилось: за-

прос создавался простым обращением к URL на веб-сервере. А теперь мы взаимодействуем с веб-сервером более сложным способом. К счастью, объект `URLConnection` выполняет за нас большую часть этой работы; нам достаточно указать, что метод запроса — `POST`, а затем указать тип отправляемых данных. Мы запрашиваем URL-соединение (объект типа `URLConnection`), вызывая метод `openConnection()` класса `URL`. Мы знаем, что работаем с протоколом HTTP, поэтому можем преобразовать данные к типу `URLConnection`, который поддерживает все необходимые возможности. Поскольку HTTP входит в набор гарантированных протоколов, это предположение безопасно. (Имеется в виду безопасность только для демонстрационных целей. Многие реальные данные считаются конфиденциальными, поэтому в отрасли рекомендуется по умолчанию использовать протокол HTTPS; подробнее об этом — в разделе «HTTPS и безопасная передача данных», с. 464.)

Затем мы вызываем метод `setRequestMethod()`, сообщая URL-соединению, что собираемся выполнить операцию `POST`. Метод `setRequestProperty()` нужен для того, чтобы указать в поле `Content-Type` HTTP-запроса соответствующий тип контента — в данном случае MIME-тип закодированных данных из формы ввода. (Это необходимо, чтобы сообщить серверу, какие данные мы ему отправляем.) Наконец, мы вызываем методы `setDoOutput()` и `setDoInput()`, чтобы через URL-соединение можно было как отправлять, так и получать потоковые данные. В результате этой комбинации действий URL-соединение знает, что мы собираемся выполнить операцию `POST`, и ожидает ответа от сервера. Затем мы вызываем метод `getOutputStream()`, чтобы получить поток вывода, и создаем объект `PrintWriter`, чтобы легко выводить закодированные данные.

Завершив отправку данных, наше приложение вызывает метод `getResponseCode()`, чтобы получить по протоколу HTTP код ответа сервера. Код ответа `HTTP_OK` означает, что все в порядке. Другие коды ответов (определенные в константах класса `URLConnection`) сообщают о всевозможных сбоях. В конце кода мы указали в комментарии, каким образом можно было бы прочитать текст ответа. Но это не обязательно: в этом простейшем случае достаточно знать, что операция `POST` завершилась успешно.

Хотя закодированные данные из форм ввода (на это указывает MIME-тип, заданный для поля `Content-Type`) встречаются чаще всего, возможны и другие виды коммуникаций. Мы могли бы использовать потоки ввода и вывода, чтобы обмениваться с серверной программой данными произвольных типов. Метод `POST` протокола HTTP позволяет передавать любые данные; серверная программа просто должна знать, как их надо обрабатывать. И последнее замечание: если вы пишете приложение, которое должно декодировать данные из форм ввода, можете использовать объект `java.net.URLDecoder` для «отмены» операции, которую совершил `URLEncoder`. Не забудьте задать кодировку UTF-8 при вызове `decode()`.

URLConnection

В классе `URLConnection` есть методы для получения связанной с запросом информации. Методы `getContentType()` и `getContentEncoding()` определяют MIME-тип и кодировку ответа. Метод `getHeaderField()` запрашивает HTTP-заголовки ответа. (Это метаданные в виде пар «имя — значение», передаваемые с ответом.) Вспомогательные методы `getHeaderFieldInt()` и `getHeaderFieldDate()` позволяют получать данные полей заголовков, содержащих целые числа и даты, — они возвращают типы `int` и `long` соответственно. Для получения длины контента и последней даты его изменения есть методы `getContentLength()` и `getLastModified()`.

HTTPS и безопасная передача данных

В предыдущем примере на сервер отправлено содержимое поля с введенным паролем. Однако стандартный протокол HTTP не дает возможности зашифровать передаваемые данные, чтобы скрыть их от посторонних. К счастью, все операции GET и POST можно сделать значительно более безопасными. Для разработчика клиентской части это очень просто. Надо всего лишь при каждой возможности использовать HTTPS — защищенную форму протокола HTTP:

```
https://www.myserver.example/cgi-bin/login.cgi
```

HTTPS — это расширение стандартного протокола HTTP. При передаче данных по протоколу HTTPS обмен данными между браузером и сервером шифруется с помощью криптографических алгоритмов с открытым ключом. Для этого протокол HTTPS использует, в частности, технологию SSL (Secure Socket Layers). Встроенная поддержка HTTPS (или SSL) есть в большинстве современных браузеров и серверов. Таким образом, если ваш веб-сервер поддерживает HTTPS и правильно настроен, вы можете безопасно обмениваться данными с браузером: для этого достаточно указать протокол `https` в ваших URL-адресах. О многочисленных аспектах безопасности (например, о достоверной идентификации другой стороны соединения) можно было бы рассказывать еще долго, но для базового шифрования данных вам нужен только HTTPS. Вам даже не придется что-то специально делать в своем коде. Стандартное издание Java JRE поставляется с поддержкой HTTPS и SSL. Поддержка не только HTTP, но и HTTPS является обязательной для всех реализаций Java, начиная с версии 5.0.

Веб-приложения Java

На первых порах существования Java все приложения на базе веб-технологий следовали одной базовой парадигме: браузер выдает запрос по конкретному

URL; сервер в ответ генерирует HTML-страницу; затем действия пользователя направляют браузер к какой-то следующей странице. В этой схеме почти вся работа выполняется на стороне сервера. Это вроде бы логично, потому что данные и службы размещаются именно на сервере. Недостаток такой модели приложений состоит в том, что она изначально ограничена потерей скорости отклика, непрерывности взаимодействия и состояний при загрузке новых страниц в браузере. Трудно сделать веб-приложение таким же быстрым и целостным, как десктопное приложение, если пользователю приходится проходить через серию отдельных страниц. К тому же возникают технические сложности с сохранением данных приложения при переходах между этими страницами. Ведь браузеры создавались не для приложений, а для просмотра документов.

Однако за последние годы в области разработки веб-приложений произошли большие изменения. С развитием стандартов HTML и JavaScript стало возможно (и даже обычно) писать приложения, в которых большая часть пользовательского интерфейса и логики реализуется на стороне клиента, а клиент обращается к серверу с фоновыми вызовами — за данными и сервисами. В этой парадигме сервер фактически возвращает всего одну страницу HTML, которая содержит ссылки на код JavaScript и CSS, а также на другие ресурсы, необходимые для формирования интерфейса приложения. С этой страницы в игру вступает основной код JavaScript; он оперирует с ее элементами или динамически создает новые элементы средствами HTML DOM (Document Object Model), чтобы пользовательский интерфейс работал. JavaScript также формирует асинхронные (фоновые) вызовы к серверу для загрузки данных и активизации сервисов. Когда-то результаты возвращались в формате XML, что привело к появлению термина *AJAX (Asynchronous JavaScript and XML)* для взаимодействий такого рода. Этот термин все еще в ходу, хотя в наши дни формат *JSON (JavaScript Object Notation)* намного популярнее XML и сейчас повсюду применяются многочисленные асинхронные библиотеки JavaScript. Поскольку в описаниях всех библиотек встречается «асинхронный JavaScript», разработчики (и менеджеры по подбору персонала) обычно говорят о конкретных используемых ими библиотеках и фреймворках, например React или Angular.

Эта новая модель упрощает разработку веб-приложений и расширяет возможности программистов. Клиенту уже не приходится работать в одностраничном режиме «запрос — ответ», где туда-сюда передаются представления и запросы. Клиент приближается к функциональности десктопного приложения: он быстрее реагирует на действия пользователя и управляет удаленными данными и сервисами без прерывания работы пользователя.

До этого момента мы использовали термин «*веб-приложение*» в обобщенном смысле, подразумевая любое браузерное приложение, размещенное на веб-сервере, будь то отдельная страница или совокупность многих страниц. Теперь мы определим этот термин точнее. В контексте API сервлетов Java типичное веб-приложение представляет собой совокупность сервлетов и веб-служб Java, которые поддер-

живают классы Java, обрабатывают контент (в том числе HTML-документы, Java Server Pages, изображения и другой мультимедийный контент) и конфигурационные данные. Для развертывания (то есть для инсталляции на веб-сервере) веб-приложение упаковывается в файл формата WAR. Файлы WAR будут подробно рассмотрены далее, а пока достаточно сказать, что это обычные архивы JAR, содержащие все файлы приложения вместе с информацией, необходимой для развертывания. Стандартизация файлов WAR означает не только портируемость кода Java, но и стандартизацию процесса развертывания приложений.

Во многих архивах WAR центральное место занимает файл `web.xml`. Это конфигурационный файл в формате XML, который описывает, какие сервлеты должны быть развернуты, их имена и URL-пути, их параметры инициализации и другие сведения, включая требования безопасности и аутентификации. Однако за последние годы файл `web.xml` стал необязательным во многих приложениях из-за появления *аннотаций* Java, занявших его место. В большинстве случаев теперь можно развертывать сервлеты и веб-службы Java простым включением аннотаций с необходимой информацией и упаковкой их в файл WAR. Возможно и объединение этих двух подходов. Эту тему мы рассмотрим далее в этой главе.

Веб-приложения работают в строго определенной исполнительной среде. Каждое веб-приложение имеет собственный «корневой» путь на веб-сервере; это означает, что все URL-адреса, ссылающиеся на свои сервлеты и файлы, начинаются с общего уникального префикса (например, `http://www.oreilly.com/someapplication`). Сервлеты каждого веб-приложения изолируются от сервлетов других веб-приложений. Никакое веб-приложение не может напрямую обращаться к файлам других веб-приложений (хотя можно разрешить это делать через веб-сервер). И у каждого веб-приложения есть собственный *контекст сервлетов*. Это понятие мы вскоре рассмотрим подробнее, но если в двух словах — это общая область, через которую сервлеты одного приложения могут обмениваться информацией и получать ресурсы от среды. Полная изоляция между веб-приложениями нужна для их динамического развертывания и обновления (как принято в современных бизнес-системах), а также для решения проблем безопасности и надежности. Каждое веб-приложение должно быть специализированным, относительно полным, не рассчитанным на жесткое связывание с другими. И хотя нет причин, которые не дают веб-приложениям взаимодействовать на высоком уровне, лучше всего объединять логику работы нескольких приложений с помощью веб-служб, которые будут рассмотрены далее в этой главе.

Жизненный цикл сервлета

Теперь перейдем к API сервлетов и приступим к их созданию. Пробелы будут восполнены далее, когда разные части API и структура файлов WAR будут рассматриваться более подробно. API сервлетов очень прост. В базовом клас-

се `Servlet` есть три метода жизненного цикла: `init()`, `service()` и `destroy()`, а также методы для получения параметров конфигурации и ресурсов сервлетов. Впрочем, эти методы далеко не всегда используются напрямую. Обычно разработчики предпочитают методы `doGet()` и `doPost()` из subclasses `HttpServlet`, а к общим ресурсам обращаются через контекст сервлета (об этом — чуть позже).

Как правило, для каждого контейнера создается только один экземпляр каждого развернутого класса сервлета. Точнее говоря, создается один экземпляр на каждую запись о сервлете, которая есть в файле `web.xml`. (Развертывание сервлетов подробнее рассмотрено в разделе «Контейнеры сервлетов», с. 479.) Раньше было одно исключение из этого правила — для специального типа однопоточных сервлетов `SingleThreadModel`. Но начиная с API сервлетов версии 2.4, этот тип объявлен устаревшим.

По умолчанию сервлеты должны обрабатывать запросы в многопоточном режиме; иначе говоря, сервисный метод сервлета (`service()`) может вызываться многими потоками одновременно. Это означает, что данные конкретных запросов и конкретных клиентов нельзя хранить в переменных экземпляра в объекте сервлета. (Конечно, там можно хранить общие данные, относящиеся к работе сервлета, если они не меняются при запросах.) Данные о состоянии конкретного клиента должны храниться в объекте клиентского *сеанса* (клиентской *сессии*) на сервере или в cookie-файлах на стороне клиента, сохраняющихся между клиентскими запросами. Мы еще вернемся к этой теме.

Метод `service()` сервлета получает два параметра: *объект запроса* к сервлету и *объект ответа* сервлета. Они предоставляют средства для чтения клиентского запроса и генерирования вывода. Мы покажем работу с ними (а вернее, с их версиями из subclasses `HttpServlet`) в приведенных ниже примерах кода.

Сервлеты

Из всех пакетов для нас основной интерес представляет пакет `javax.servlet.http`. Он содержит специфический для сервлетов API, предназначенный для обработки запросов HTTP к веб-серверам. Теоретически сервлеты можно писать и для других протоколов, но на практике этим никто не занимается, поэтому мы будем обсуждать сервлеты только в контексте протокола HTTP.

Обратите внимание на префикс `javax`: он уже встречался нам при рассмотрении пакетов `Swing`. Несомненно, API сервлетов является важной частью Java, но он не входит в базовый инструментарий разработчика. Вам придется загрузить отдельную библиотеку `servlet-api.jar` от стороннего поставщика. Apache предоставляет эталонную реализацию API сервлетов. Дополнительные инструкции по установке этой библиотеки и по ее использованию в командной строке и в IntelliJ IDEA приведены в разделе «Где загрузить примеры кода», с. 497.

Важнейшим инструментом пакета `javax.servlet.http` является базовый класс `HttpServlet`. Это абстрактный сервлет, в котором реализован ряд основных задач обработки запросов HTTP. В частности, он переопределяет общий запрос `service()` и разделяет его на методы, соответствующие операциям HTTP, включая `doGet()`, `doPost()`, `doPut()` и `doDelete()`. Метод `service()` анализирует запрос, проверяет вид запроса и передает его одному из этих четырех методов. Вы можете переопределить один или несколько из них, чтобы реализовать нужную вам логику работы протокола.

Методы `doGet()` и `doPost()` соответствуют стандартным методам `GET` и `POST` из протокола HTTP. Напомним, что `GET` — это обычный запрос на получение файла или документа по заданному URL, а `POST` — метод протокола, который используется клиентом для отправки произвольного объема данных на сервер. Методом `POST` на сервер передаются данные из форм ввода HTML, а также данные большинства веб-служб.

Еще два метода, `doPut()` и `doDelete()`, соответствуют той части протокола HTTP, которая широко распространена в веб-приложениях на основе API в стиле REST (REpresentational State Transfer). Эти методы нужны, чтобы отправлять на сервер и удалять файлы и другие сущности, например записи баз данных. По функциональности `doPut()` напоминает `POST`, но с тем отличием, что `PUT` логически замещает элемент, на который указывает URL-адрес, тогда как `POST` просто отправляет данные по этому адресу. Метод `doDelete()`, наоборот, удаляет с сервера указанный элемент.

В классе `HttpServlet` также есть три других метода, относящихся к протоколу HTTP: `doHead()`, `doTrace()` и `doOptions()`. Обычно переопределять эти методы не требуется. Метод `doHead()` выполняет запрос HTTP `HEAD`, который запрашивает заголовки запроса `GET` без тела. Это делается примитивным способом: сначала выполняется запрос `GET`, а затем отправляются только заголовки. Возможно, в каких-то особых случаях вы захотите переопределить `doHead()` вашей собственной, более эффективной реализацией, если вам понадобится оптимизация. Методы `doTrace()` и `doOptions()` реализуют другие функции протокола HTTP, которые обеспечивают отладку и упрощают взаимодействие клиента и сервера. Как правило, их не приходится переопределять.

Наряду с `HttpServlet` пакет `javax.servlet.http` включает субклассы объектов `ServletRequest` и `ServletResponse`, а также `HttpServletRequest` и `HttpServletResponse`. Эти субклассы предоставляют, соответственно, потоки ввода и вывода, необходимые для чтения и записи данных на стороне клиента. Они также предоставляют API для чтения и записи информации заголовков HTTP и, как будет показано ниже, информации клиентских сеансов. Вместо формального описания мы продемонстрируем их в примерах кода. Как обычно, начнем с самого простого примера из всех возможных.

Сервлет `HelloClient`

Ниже приведена наша версия программы «Hello, World» в виде сервлета `HelloClient`:

```
@WebServlet(urlPatterns={"/hello"})
public class HelloClient extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html"); // Эта строка должна быть
                                                // первой в блоке
        PrintWriter out = response.getWriter();
        out.println(
            "<html><head><title>Hello Client!</title></head><body>"
            + "<h1>Hello Client!</h1>"
            + "</body></html>" );
    }
}
```

Если вы хотите сразу опробовать этот сервлет, переходите к разделу «Контейнеры сервлетов», с. 479; там описан процесс развертывания сервлета. Мы включили в класс аннотацию `WebServlet`, поэтому сервлету для развертывания не нужен файл `web.xml`. Все, что вам надо сделать, — это упаковать файл класса в конкретный каталог внутри архива WAR (обычного ZIP-файла) и поместить этот архив в каталог, отслеживаемый сервером Tomcat. А мы сейчас сосредоточимся на коде этого сервлета, который предельно прост. Все примеры кода из этой части книги доступны во втором репозитории на GitHub (<https://github.com/10y/learnjava5e-web>). О том, как загрузить и настроить IntelliJ IDEA для работы с соответствующей библиотекой сервлетов, рассказано в разделе «Загрузка примеров кода», с. 64.

Рассмотрим код. Класс `HelloClient` расширяет базовый класс `HttpServlet` и переопределяет метод `doGet()` для обработки простых запросов. В данном случае мы будем реагировать на любой запрос GET отправкой однострочного документа HTML с текстом «Hello Client!». Сначала мы сообщаем контейнеру тип генерируемого ответа, для чего вызываем метод `setContentType()` из объекта типа `HttpServletResponse`. Поскольку ответом будет HTML-документ, мы задаем MIME-тип «text/html». Затем мы получаем поток вывода методом `getWriter()` и выводим в него сообщение. Явно закрывать поток не обязательно. Управление потоками вывода будет рассмотрено далее в этой главе.

Исключения `ServletException`

Метод `doGet()` в нашем примере сервлета объявляет, что он может выдавать исключение `ServletException`. Каждый сервисный метод API сервлетов может

выдавать исключение `ServletException`, чтобы сообщить о неудачной попытке запроса. Объект `ServletException` может конструироваться со строковым сообщением и необязательным параметром типа `Throwable`, указывающим на любое соответствующее исключение, представляющее первоначальную причину проблемы:

```
throw new ServletException( "utter failure", someException );
```

По умолчанию веб-сервер самостоятельно формирует то сообщение, которое видит пользователь при исключении `ServletException`. Во многих случаях разработчику доступен режим отладки (*development mode*), в котором выводится информация о каждом исключении, в том числе трассировка стека. Многие контейнеры сервлетов (например, Tomcat) позволяют разработчику формировать собственные страницы с сообщениями об ошибках, но эта тема выходит за рамки книги.

В качестве альтернативы сервлет может выдать исключение класса `UnavailableException` (субкласса `ServletException`), означающее, что он не способен обрабатывать запросы. Это исключение выдается как в случае постоянной неработоспособности, так и в случае временной приостановки работы — с указанием соответствующего периода в секундах.

Тип контента

Итак, перед получением потока вывода мы должны были сообщить, какие данные будем отправлять. Для этого мы вызвали метод `setContentType()` и указали тип `text/html`, то есть MIME-тип, соответствующий документам HTML. Но в общем случае сервлет может генерировать данные любого типа, включая аудиофайлы, видеофайлы и всевозможные текстовые или двоичные файлы. Если бы мы решили написать универсальный сервлет `FileServlet`, предоставляющий клиентам файлы, как обычный веб-сервер, то он мог бы анализировать расширение каждого файла и определять MIME-тип по расширению или путем прямого анализа содержимого файла. (Это хорошее применение для метода `probeConentType()` из класса `java.nio.file.Files`.) Для записи двоичных данных можно вызвать метод `getOutputStream()`, чтобы получить `OutputStream` вместо `Writer`.

Отправляя ответ по протоколу HTTP, сервер указывает тип контента в заголовке `Content-Type:`, чтобы еще до начала загрузки клиент знал, какой файл он получит. Это позволяет браузеру отобразить диалоговое окно сохранения файла при щелчке на ZIP-архиве или исполняемой программе. Если строка, задающая тип контента, имеет полную форму с указанием кодировки символов (например, `text/html; charset=UTF-8`), то сервлет назначит эту же кодировку потоку вывода `PrintWriter`. Это означает, что вы всегда должны вызывать метод `setContentType()` перед вызовом `getWriter()`. Кроме того, вы можете указать кодировку символов отдельно, вызовом метода `setCharacterEncoding()` из объекта ответа сервлета.

Ответ сервлета

Кроме предоставления потока вывода для передачи контента клиенту, объект `HttpServletResponse` предоставляет методы для управления другими частями ответа HTTP (включая заголовки, коды ошибок и перенаправления) и буферизацией контейнера сервлета.

Заголовки HTTP — это метаданные в форме пар «имя — значение», передаваемые с ответом. Вы можете добавлять в ответ стандартные и нестандартные заголовки при помощи методов `setHeader()` и `addHeader()` (заголовки могут содержать несколько значений). Кроме того, вы можете заполнять заголовки целочисленными значениями и датами:

```
response.setIntHeader( "MagicNumber", 42 );
response.setDateHeader( "CurrentTime", System.currentTimeMillis() );
```

Когда вы записываете данные на сторону клиента, контейнер сервлета автоматически присваивает коду ответа HTTP значение 200, что означает успешное выполнение («ОК»). При помощи метода `sendError()` можно отправлять другие коды ответов HTTP. `HttpServletResponse` содержит заранее определенные константы для всех стандартных кодов. Несколько самых распространенных кодов:

```
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
```

Как только вы генерируете ошибку методом `sendError()`, ответ завершается, и после этого вы уже не сможете передать значимый контент на сторону клиента. Впрочем, вы можете указывать на некоторые ошибки, выводя на стороне клиента короткие сообщения.

Перенаправление HTTP — особый вид ответа, который дает указание клиенту (то есть браузеру) перейти по другому URL-адресу. Обычно этот переход происходит быстро и без какого-либо взаимодействия с пользователем. Для отправки перенаправления служит метод `sendRedirect()`:

```
response.sendRedirect("http://www.oreilly.com/");
```

Говоря об ответах сервера, мы должны сказать несколько слов и о буферизации. Многие ответы сначала помещаются во внутренний буфер контейнера сервлетов — до того момента, когда завершится сервисный метод сервлета или будет достигнут заранее заданный максимальный размер буфера. Это позволяет контейнеру задать заголовок HTTP `content-length` автоматически, чтобы

сообщить клиенту, какой объем данных ему следует ожидать. Для управления размером буфера служит метод `setBufferSize()` с передачей размера в байтах. Вы даже можете очистить буфер и начать его заполнение заново, если данные еще не были записаны на сторону клиента. Чтобы очистить буфер, вызовите метод `isCommitted()` для проверки того, были ли отправлены какие-либо данные, а затем метод `resetBuffer()` для сброса буфера, если данные еще не отправлены. Если вы отправляете большой объем данных, лучше явно задать длину контента методом `setContentLength()`.

Параметры сервлетов

В нашем первом примере кода мы продемонстрировали реакцию на очень простой запрос. Разумеется, чтобы сделать что-то более полезное, мы должны сначала получить какую-то информацию от клиента. К счастью, этим занимается сам сервлет: он интерпретирует закодированные данные из форм ввода, переданные клиентом в запросах `GET` и `POST`, и передает их нам с помощью простого метода `getParameter()` из объекта запроса к сервлету.

`GET`, `POST` и «расширенный путь»

Вы уже знаете про два обычных способа передачи информации от браузера сервлету или программе `CGI`. Основной способ — `POST` (клиент кодирует информацию и отправляет ее в виде потока данных программе, которая ее декодирует). С помощью `POST` на сервер передаются и данные из форм ввода, и большие объемы любых других данных, в том числе файлы. Второй способ — встраивание данных в `URL`-адрес. Как правило, для этого выполняется запрос в стиле `GET`: браузер кодирует данные и добавляет их в конец строки `URL`-адреса в качестве параметров. Затем сервер декодирует их и передает приложению.

Напомним, что кодирование в стиле `GET` присоединяет параметры к `URL`-адресу в виде пар «имя — значение»; перед первым параметром ставится вопросительный знак (`?`), а каждый следующий параметр отделяется символом `&`. Вся строка должна быть *URL-кодированной* (*URL-encoded*), то есть все специальные символы в ней (включая пробелы, а также символы `?` и `&`) должны заменяться специальными последовательностями.

Есть еще один способ передачи данных в `URL`-адресе: *расширенный путь* (*extra path*). Он достаточно прост: сервер находит указанный в `URL`-адресе сервлет (или программу `CGI`), а затем берет из `URL`-адреса все последующие компоненты пути и передает их этому сервлету. Для примера возьмем два адреса:

```
http://www.myserver.example/servlets/MyServlet
http://www.myserver.example/servlets/MyServlet/foo/bar
```


Допустим, сервер связывает первый адрес с сервлетом `MyServlet`. При получении второго адреса сервер также вызывает `MyServlet`, но считает `/foo/bar` «расширенным путем», который может быть получен методом `getPathInfo()` из объекта запроса к сервлету. Этот прием помогает составлять понятные человеку, осмысленные URL-адреса — они удобны при работе с контентом, состоящим из отдельных документов.

Для кодирования данных из форм ввода HTML на стороне клиента можно использовать как `GET`, так и `POST`. Укажите, соответственно, или `get`, или `post` в атрибуте `action` тега `form`. Браузер выполнит кодирование, а сервлет на стороне сервера — декодирование.

Клиент, передавая сервлету данные из форм ввода, указывает для них следующий тип: `application/x-www-form-urlencoded`. API сервлетов автоматически разбирает такие данные и предоставляет доступ к ним через метод `getParameter()`. Но если вы не вызовете метод `getParameter()`, то данные все равно останутся доступными в неразобранном виде в потоке ввода и сервлет может их прочитать напрямую.

GET или POST: что лучше?

С точки зрения пользователя, основное различие между `GET` и `POST` заключается в том, что информация `GET` видна в закодированном URL-адресе, который отображается в браузере. Иногда это бывает удобно: пользователь может копировать URL-адреса (например, результаты поиска), отправлять их друзьям по почте и сохранять в закладках на будущее. Информация `POST` не видна пользователю и удаляется после отправки на сервер. Так и было задумано при разработке протокола HTTP: `GET` и `POST` должны отличаться по смыслу. Существует правило: результат запроса `GET` не должен иметь побочных эффектов, то есть такой запрос не должен заставлять сервер выполнять операции с долгосрочным эффектом (например, оформление покупки в интернет-магазине). Считается, что этим должны заниматься запросы `POST`. Вот почему ваш браузер предупреждает о повторной отправке данных из форм ввода, когда вы нажимаете кнопку перезагрузки на странице, полученной в результате отправки формы.

Стиль «расширенный путь» хорош для сервлетов, которые работают с файловыми структурами или обрабатывают диапазоны URL-адресов, составленных в понятной человеку форме. Чаще всего этот способ передачи данных используют для таких URL-адресов, которые привычно выглядят и легко запоминаются пользователями.

Сервлет ShowParameters

Наш первый сервлет почти ничего не делал. Но в следующем примере сервлет будет выводить значения любых полученных им параметров. Начнем с обработки

запросов GET, а затем внесем несколько простых изменений, чтобы обрабатывались еще и запросы POST. Код выглядит так:

```
import java.io.*;
import javax.servlet.http.*;
import java.util.*;

@WebServlet(urlPatterns={"/showParameter"})
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>Show Parameters</title></head><body>"
            + "<h1>Parameters</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( );
    }
}
```

Как и в первом примере, мы переопределяем метод `doGet()`. Запрос делегируется вспомогательному методу `showRequestParameters()`, который мы создали; он перечисляет параметры с помощью метода `getParameterMap()` из объекта запроса. Этот метод возвращает карту (ассоциативный массив), связывающую имена параметров с их значениями. Заметим, что параметр может иметь несколько значений, если они повторяются в запросе от клиента, поэтому карта содержит `String []`. Результатом работы сервлета будет аккуратный список, так как перед каждым выводимым параметром есть тег ``.

В таком виде сервлет будет отвечать на любой URL с запросом GET. Теперь доработаем код: добавим форму ввода данных и поддержку запросов POST. Для их

обработки мы переопределим метод `doPost()`. Наша реализация метода `doPost()` могла бы вызывать метод `showRequestParameters()`, но есть еще более простой способ. API позволяет интерпретировать запросы GET и POST как взаимозаменяемые, так как ядро сервлетов обеспечивает декодирование параметров запроса. Поэтому проще всего делегировать операцию `doPost()` методу `doGet()`.

Итак, добавьте в сервлет следующий метод:

```
public void doPost( HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    doGet( request, response );
}
```

А теперь займемся формой ввода HTML. Через нее пользователь будет вводить параметры и передавать их сервлету. Добавьте следующую строку в метод `showRequestParameters()` непосредственно перед вызовом метода `out.close()`:

```
out.println("</ul><p><form method=\"POST\" action=\""
    + request.getRequestURI() + "\">"
    + "Field 1 <input name=\"Field 1\" size=20><br>"
    + "Field 2 <input name=\"Field 2\" size=20><br>"
    + "<br><input type=\"submit\" value=\"Submit\"></form>"
);
```

Атрибут `action` формы ввода содержит URL нашего сервлета, который таким образом получает введенные пользователем данные. Метод `getRequestURI()` определяет местонахождение сервлета. В атрибуте `method` задается операция POST, но вы можете попробовать заменить ее на GET, чтобы увидеть оба стиля в действии.

До настоящего момента мы не сделали ничего впечатляющего. В следующем примере мы расширим возможности приложения, добавив пользовательский сеанс для хранения клиентских данных между запросами.

Управление сеансами пользователей

Одна из самых ценных возможностей API сервлетов — простой механизм управления пользовательскими сеансами (сессиями). Под сеансом мы имеем в виду, что сервлет хранит данные конкретного пользователя в то время, пока пользователь переходит между страницами и выполняет при этом всевозможные действия. Это также называют сохранением состояния (*maintaining state*). Целостность пользовательских данных в пределах всей серии веб-страниц необходима во многих приложениях, например при идентификации по логину и паролю, а также при накоплении покупок в корзине интернет-магазина. В каком-то смысле сеансовые данные служат вместо данных экземпляра в объ-

екте вашего сервлета, так как сеансовые данные тоже сохраняются между вызовами сервисных методов. Без этого механизма ваш сервлет не смог бы узнать, что два запроса поступили от одного и того же пользователя.

Все сеансы отслеживает контейнер сервлетов, и вам не придется этим заниматься в обычных приложениях. Для идентификации сеансов есть два способа: куки-файлы на стороне клиента и замена URL. *Куки (cookies)* — это стандартный механизм протокола HTTP, позволяющий серверу сохранить состояние во взаимодействии с браузером пользователя. Каждая запись в куки-файле — это атрибут «имя — значение», который создается сервером, хранится на клиентском компьютере и возвращается им всякий раз, когда браузер обращается к определенной группе URL-адресов на указанном сервере. С помощью куки-файлов можно отслеживать единичную сессию, а также множество повторных сессий.

При *замене URL (URL rewriting)* в URL-адрес добавляется идентификатор сеанса — или закодированный в стиле GET, или в виде расширенного пути. Слово «замена» означает, что перед отправкой клиенту сервер модифицирует URL-адрес, а перед возвратом сервлету удаляет эту дополнительную информацию. Для поддержки этого способа сервлет должен кодировать каждый URL-адрес на страницах в пределах сеанса (в том числе каждую ссылку, которая может возвращать к этим страницам). Такое кодирование — это дополнительная операция, выполняемая соответствующим методом объекта ответа (`HttpServletResponse`). Вы должны разрешить серверу замену URL-адресов, если хотите, чтобы ваше приложение работало с браузерами, в которых куки-файлы отключены или не поддерживаются. (Впрочем, многие сайты принципиально не работают с такими браузерами.)

Программист сервлетов получает доступ к информации состояния через объект `HttpSession`, действующий как хеш-таблица для хранения любых объектов, которые должны храниться на протяжении сеанса. Эти объекты остаются на стороне сервера, а клиенту передается специальный идентификатор: с помощью куки-файла или замены URL-адреса. Когда клиент делает запрос, этот идентификатор обозначает конкретный сеанс, который ассоциируется с сервлетом.

Сервлет ShowSession

Ниже приведен код простого сервлета, показывающий, как хранить строковую информацию для идентификации сеанса:

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import java.util.Enumeration;

@WebServlet(urlPatterns={"/showSession"})
```

```

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>Show Session</title></head><body>");

        if ( clear )
            out.println("<h1>Session Cleared:</h1>");
        else {
            out.println("<h1>In this session:</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() + "\">"
            + "Name: <input name=\"Name\" size=20><br>"
            + "Value: <input name=\"Value\" size=20><br>"
            + "<br><input type=\"submit\" value=\"Submit\">"
            + "<input type=\"submit\" name=\"clear\" value=\"Clear\"></form>"
            );
    }
}

```

При вызове этого сервлета в браузере появляется форма, предлагающая ввести имя и значение. Строка значения сохраняется в объекте сеанса под заданным именем. При каждом вызове сервлет выводит список всех элементов данных, связанных с сеансом. Вы увидите, что количество сеансовых данных увеличивается с добавлением каждого элемента (в данном случае — до перезапуска браузера или сервера).

По принципу работы этот пример очень похож на сервлет `ShowParameters`. Метод `doGet()` генерирует форму ввода, которая отправляет данные сервлету методом `POST`. Мы переопределяем метод `doPost()`, чтобы он возвращал управление методу `doGet()`, который обрабатывает все запросы. В методе `doGet()` мы пытаемся получить объект сеанса пользователя из объекта запроса методом `getSession()`. Объект `HttpSession`, предоставляемый запросом, работает как хеш-таблица. В нем есть метод `setAttribute()`, получающий имя типа `String` и второй аргумент типа `Object`; а также соответствующий метод `getAttribute()`. Метод `getAttributeNames()` перечисляет и выводит все значения, которые в данный момент хранятся в сеансе.

Если сеанс еще не существует, то метод `getSession()` создает его по умолчанию. Если вы хотите проверить существование сеанса или явно управлять его созданием, вызовите перегруженную версию метода `getSession(false)`, которая не создает новый сеанс автоматически и возвращает `null`, если сеанс не существует. Также можно проверить, был ли сеанс только что создан, при помощи метода `isNew()`. Для немедленной очистки сеанса используется метод `invalidate()`. После вызова `invalidate()` для сеанса дальнейшие обращения к нему невозможны, поэтому мы устанавливаем флаг и выводим сообщение об очистке сеанса. Сеансы могут становиться недействительными, когда достигнут лимит времени. Вы можете задавать лимит времени сеанса (`session timeout`) на программном уровне, или в сервере приложений, или в файле `web.xml` (значение параметра `session-timeout` в разделе `session-config`). Для приложения превышение лимита времени означает, что сеанс не существует или он будет создан при очередном запросе. Каждый сеанс пользователя связан только с одним конкретным приложением и не может использоваться другими.

Ранее мы упоминали о дополнительной операции, необходимой для поддержки замены URL-адреса, то есть для браузеров, не поддерживающих куки-файлы. Она заключается в том, что каждый URL-адрес, генерируемый сервлетом в контенте сессии, надо предварительно модифицировать методом `encodeURL()` из объекта ответа (`HttpServletResponse`). Этот метод получает строку с URL-адресом и возвращает модифицированную строку только в том случае, если нужна замена URL-адреса. Обычно при доступности куки-файлов он возвращает ту же самую строку. В предыдущем примере мы могли бы закодировать URL-адрес формы ввода, полученный от `getRequestURI()`, прежде чем передавать его клиенту. Тогда сервлет подошел бы и браузерам без куки-файлов.

Контейнеры сервлетов

Наконец-то настало время выполнить весь код примеров! Есть много инструментов, предназначенных для развертывания серверов, — они называются *контейнерами*. Ни OpenJDK, ни официальный Oracle JDK не содержат встроенного контейнера сервлетов. Но интернет-сервисы (такие, как AWS¹) предоставляют относительно быстрые и недорогие контейнеры, которые делают ваши сервлеты доступными для всего мира. Впрочем, для разработки вы, конечно, предпочтете локальную среду, находящуюся под вашим контролем, которую можно свободно изменять и перезапускать в процессе изучения API сервлетов. Ее придется настраивать самостоятельно, и для этой цели хорошо подойдет «эталонная реализация» контейнера Apache Tomcat. Мы будем устанавливать версию 9, но более старые версии тоже поддерживают все основные возможности сервлетов, которые мы рассмотрели.

Как упоминалось ранее, файл WAR — это архив, содержащий все части веб-приложения: файлы классов Java для сервлетов и веб-служб, JSP-файлы, страницы HTML, графику и другие ресурсы. Файл WAR является обычным архивом JAR (который, в свою очередь, является особой разновидностью архива ZIP) с заданными каталогами для кода Java и с одним конфигурационным файлом `web.xml` — этот файл сообщает серверу приложения, что и как надо запускать. Файлы WAR всегда имеют расширение `.war`, но для их создания и чтения можно использовать стандартную утилиту `jar`.

Содержимое типичного WAR-файла может выглядеть примерно так (при выводе в программе `jar`):

```
$ 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
```

При развертывании имя WAR-файла по умолчанию становится корневым путем веб-приложения, в данном случае `shoppingcart`. Таким образом, базовый URL-

¹ Amazon Web Services — один из крупнейших провайдеров с разными предложениями, от бесплатных пробных периодов до сервисов корпоративного уровня. Есть очень много других вариантов хостинга Java-приложений, включая Heroku и Google App Engine (последний не является контейнером сервлетов как таковым, но все же позволяет применить ваши навыки программирования на Java в интернете).

адрес веб-приложения при развертывании на `http://www.oreilly.com` принял бы вид `http://www.oreilly.com/shoppingcart/`, а все ссылки на его документы, графические файлы и сервлеты начинались бы с этого пути. Верхний уровень файла WAR становится корневым каталогом документов (базовым каталогом) для файлов, предоставляемых клиентам. Наш файл `index.html` расположен по базовому URL-адресу, как упоминалось ранее. Ссылка на изображение `happybunny.gif` имела бы вид `http://www.oreilly.com/shoppingcart/images/happybunny.gif`.

Каталог `WEB-INF` (в верхнем регистре и с дефисом) — специальный каталог, содержащий всю информацию для развертывания и код приложения. Этот каталог защищен веб-сервером, а его содержимое остается невидимым для внешних пользователей приложения, даже если добавить `WEB-INF` к базовому URL-адресу. Классы вашего приложения могут загружать дополнительные файлы из этого каталога вызовом метода `getResource()` для контекста сервлета; это безопасное место для хранения ресурсов приложения. В каталоге `WEB-INF` также есть файл `web.xml`, о котором мы еще поговорим в следующем разделе.

Каталоги `WEB-INF/classes` и `WEB-INF/lib` содержат, соответственно, файлы классов Java и библиотеки в JAR-файлах. Каталог `WEB-INF/classes` автоматически добавляется в `classpath` веб-приложения, которому таким образом становятся доступны все файлы классов, находящиеся в этом каталоге (упакованные по обычным соглашениям о пакетах Java). После этого в `classpath` веб-приложения добавляются все JAR-файлы, расположенные в `WEB-INF/lib` (к сожалению, порядок их добавления не определен). Вы можете размещать свои классы в любом из этих каталогов. Для разрабатываемых классов удобен не требующий упаковки каталог `classes`, а каталог `lib` лучше подходит для вспомогательных классов и сторонних библиотек. Кроме этого, можно устанавливать JAR-файлы непосредственно в контейнере сервлетов, чтобы сделать их доступными для всех веб-приложений, выполняемых на этом сервере. Часто так размещают общие библиотеки, необходимые многим веб-приложениям. Впрочем, место для размещения таких библиотек не стандартизировано, и любые классы, развернутые таким образом, не будут автоматически перезагружаться после внесения в них изменений (но вообще для WAR-файлов это возможно, о чем мы поговорим далее). API сервлетов требует, чтобы каждый сервер предоставлял каталог для этих JAR-пакетов, при этом классы должны быть видимы для веб-приложений и загружаться только одним загрузчиком.

Настройка конфигурации с использованием `web.xml` и аннотаций

Файл `web.xml` — это конфигурационный файл в формате XML, в котором перечисляются сервлеты и другие предназначенные для развертывания элементы, относительные имена (URL-адреса), под которыми они будут развертываться, их параметры инициализации и развертывания (в том числе относящиеся к безо-

пасности и авторизации). На протяжении почти всей истории веб-приложений Java этот механизм конфигурирования оставался единственным. Но в API сервлетов 3.0 (Tomcat 7 и выше) появились дополнительные возможности. Большую часть конфигурирования теперь можно выполнять посредством аннотаций Java. Напомним, что в нашем первом сервлете `HelloClient` мы использовали аннотацию `@WebServlet`, чтобы объявить сервлет и указать URL-адрес его развертывания. С аннотацией можно развернуть сервлет на сервере Tomcat совсем без файла `web.xml`. Кроме того, в API сервлетов 3.0 появилась возможность процедурного развертывания сервлетов — из выполняемого кода Java.

В этом разделе мы опишем оба способа настройки конфигурации — как в формате XML, так и с аннотациями. В большинстве случаев аннотации удобнее, но XML-конфигурацию тоже надо знать как минимум по двум причинам. Прежде всего файл `web.xml` может вам понадобится, чтобы переопределить или расширить конфигурацию, жестко зафиксированную в аннотации. Он позволяет изменить конфигурацию во время развертывания без перекомпиляции классов. В общем случае XML-конфигурация имеет более высокий приоритет, чем аннотации. Вы даже можете дать серверу указание полностью игнорировать аннотации — для этого предназначен атрибут `metadata-complete` в файле `web.xml`. Кроме того, некоторые параметры конфигурации, особенно относящиеся к контейнеру сервлетов, задаются только в файле `web.xml`.

Будем считать, что вы хотя бы в общих чертах знакомы с XML, но в крайнем случае просто скопируйте последующие примеры. Начнем с простого файла `web.xml` для нашего сервлета `HelloClient`. Он выглядит так:

```
<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>
```

Элемент верхнего уровня в этом документе называется `<web-app>`. Внутри `<web-app>` могут находиться многие типы элементов, но два основных — это `<servlet>` и `<servlet-mapping>`. Тег `<servlet>` служит для объявления экземпляра сервлета и, опционально, для его инициализации и конфигурирования. Для каждого тега `<servlet>`, присутствующего в файле `web.xml`, создается один экземпляр класса сервлета.

В объявлении экземпляра сервлета тегом `<servlet>` должно быть два обязательных элемента: `<servlet-name>`, который служит дескриптором для ссылок на сервлет в других местах файла `web.xml`, и `<servlet-class>`, который

задает для сервлета имя класса Java. В этом примере сервлету присвоено имя `helloclient1`. Мы выбрали его, чтобы подчеркнуть, что при желании могли объявить другие экземпляры того же сервлета, в том числе с другими параметрами инициализации, и т. д. Классу сервлета мы, конечно, дали имя `HelloClient`. В реальном приложении ему было бы присвоено полное имя, например: `com.oreilly.servlets>HelloClient`.

В объявление сервлета также может входить один или несколько параметров инициализации, доступ к которым осуществляется методом `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>
```

Затем идет элемент `<servlet-mapping>`, назначающий экземпляру сервлета путь на веб-сервере:

```
<servlet-mapping>
  <servlet-name>helloclient1</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

Здесь сервлету назначен путь `/hello`. (При желании можно было бы добавить и другие элементы `<url-pattern>`.) Если мы дадим нашему WAR-файлу имя `learningjava.war` и развернем его на сервере `www.oreilly.com`, то полный путь к сервлету будет таким: `http://www.oreilly.com/learningjava/hello`. Напомним, что тегами `<servlet>` можно объявить несколько экземпляров сервлета. И для каждого из них можно объявить несколько элементов `<servlet-mapping>`. Например, одному экземпляру `helloclient1` можно назначить основной путь `/hello` и резервный путь `/hola`. Элемент `<url-pattern>` дает много возможностей назначать подходящие URL-адреса, которые должны соответствовать сервлету. Мы подробнее рассмотрим это в следующем разделе.

Хотя приведенный ранее пример `web.xml` будет работать на некоторых серверах приложений, он неполон с технической точки зрения, так как в нем не указаны используемые версии формата XML и стандарта файла `web.xml`. Для полного соответствия стандартам добавьте в файл такую строку:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Начиная с API сервлетов 2.5 информация о версии в файле `web.xml` должна соответствовать спецификации XML Schema. Для этого надо вставить в элемент `<web-app>` дополнительную информацию:

```
<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">
```

Без этой дополнительной информации приложение все равно может работать, но контейнеру сервлетов будет труднее обнаруживать ошибки в конфигурации и выдавать понятные сообщения об ошибках. Кроме того, эта информация нужна некоторым специализированным текстовым редакторам — для цветового выделения синтаксиса, автозавершения и других приятных мелочей.

Описанное выше объявление сервлета и назначение путей эквивалентны следующей однострочной аннотации:

```
@WebServlet(urlPatterns={"/hello", "/hola"})
public class HelloClient extends HttpServlet {
    ...
}
```

Здесь атрибут `urlPatterns` класса `WebServlet` позволяет назначить один или несколько путей — с таким же результатом, как при их назначении элементами `<url-pattern>` в файле `web.xml`.

URL-шаблоны

Шаблон `<url-pattern>` в предыдущем примере был простой строкой `/hello`. Это означает, что сервлет должен запускаться только по одному URL-адресу (по единственно возможному запросу), состоящему из базового URL-адреса с последующим суффиксом `/hello`. Но тег `<url-pattern>` позволяет задавать и более сложные шаблоны с групповыми символами `*`. Например, с шаблоном `/hello*` наш сервлет будет запускаться по таким URL-адресам, как `http://www.oreilly.com/learningjava/helloworld` или `.../helloworld`. Групповые символы можно указывать даже перед расширениями; например, шаблон `*.html` или `*.foo` будет запускать сервлет по любым URL-адресам, которые заканчиваются этими символами.

Групповые символы в шаблонах могут привести к неоднозначности совпадений. Возьмем шаблоны URL, завершающиеся суффиксами `/scooby*` и `/scoobydoo*`. Какой из них будет выбран для URL-адреса, завершающегося символами `.../scoobydoobiedoo`? А если будут и другие соответствия, например, из-за расширения? Подобные конфликты разрешаются по следующим правилам.

Сначала выбираются точные совпадения. Для URL-адреса `.../hello` в нашем примере будет выбран шаблон `/hello`, несмотря на любые дополнительные шаблоны с групповыми символами, например `/hello*`. Если точных совпадений нет, то контейнер ищет самое длинное совпадение с начала строки. То есть для

URL-адреса `.../scoobydoobiedoo` контейнер выберет шаблон `/scoobydoo*` (а не `/scooby*`), потому что это совпадение длиннее и предположительно точнее. Если совпадений снова нет, то контейнер проверяет шаблоны с групповыми символами в суффиксах. На этом этапе процесса для URL-адреса, завершающегося на `.foo`, будет выбран шаблон `*.foo`. Наконец, при полном отсутствии совпадений контейнер обратится к универсальному шаблону по умолчанию: `/*`. Сервлет, связанный с шаблоном `/*`, «подберет» любой запрос, если все предыдущие варианты не подошли. А в том случае, если нет даже этой связи по умолчанию, обработка запроса будет прервана, а клиент получит сообщение «404 Not Found».

Развертывание сервлета HelloClient

После того как сервлет `HelloClient` будет развернут, вы сможете легко добавлять в WAR-файл приведенные в этой главе примеры. В этом разделе мы покажем, как создать WAR-файл вручную. Конечно, есть ряд инструментов, упрощающих создание и пополнение WAR-файлов, но и вручную это делается довольно просто; к тому же вы будете уделять больше внимания содержимому вашего архива.

Прежде всего создайте каталоги `WEB-INF` и `WEB-INF/classes`. Если вы выбрали конфигурацию с файлом `web.xml`, поместите его в каталог `WEB-INF`. (Не забывайте, что файл `web.xml` необязателен, если вы используете аннотацию `WebServlet` с контейнером Tomcat версии 7 и выше.) Затем поместите файл `HelloClient.class` в каталог `WEB-INF/classes`. Создайте архив `learningjava.war` следующей командой `jar` (каталог `WEB-INF` будет на верхнем уровне архива):

```
$ jar cvf learningjava.war WEB-INF
```

Далее вы можете добавлять в этот WAR-файл документы и другие ресурсы, указывая их имена после каталога `WEB-INF`. Для проверки содержимого WAR-файла воспользуйтесь командой `jar`:

```
$ jar tvf learningjava.war
document1.html
WEB-INF/web.xml
WEB-INF/classes/HelloClient.class
```

Теперь остается лишь поместить WAR-файл в нужный каталог на вашем сервере. Загрузите и установите контейнер сервлетов Apache Tomcat, если еще не сделали этого. На официальном сайте Apache (<https://tomcat.apache.org>) вы найдете последнюю версию Tomcat и полезную документацию.

Предназначенный для WAR-файлов каталог `webapps` находится в том каталоге, куда вы установили Tomcat. Поместите ваш WAR-файл в `webapps` и затем запустите сервер. Если в конфигурации Tomcat установлен номер порта по умолчанию, то вы сможете обращаться к сервлету `HelloClient` по одному из двух следующих URL-адресов: `http://localhost:8080/learningjava/hello` или

`http://<ваш_сервер>:8080/learningjava/hello`, где `<ваш_сервер>` — это имя или IP-адрес вашего сервера. Если у вас возникнут трудности, поищите информацию об ошибках в каталоге `logs`, который тоже находится в том каталоге, куда вы установили Tomcat.

Перезагрузка веб-приложений

Во всех контейнерах сервлетов есть средства перезагрузки WAR-файлов; многие контейнеры поддерживают перезагрузку отдельных классов сервлетов после их модификации. Согласно спецификации сервлетов, возможность перезагрузки WAR-файлов обязательна, она особенно полезна в процессе разработки. Но конкретный способ перезагрузки веб-приложений зависит от сервера. Как правило, достаточно поместить новый WAR-файл вместо прежнего в тот же каталог (например, в каталог `webapps` контейнера Tomcat). Обнаружив эту замену файла, контейнер завершит работу прежнего приложения и развернет вместо него новое. Этот механизм работает в Tomcat при установленном атрибуте `autoDeploy` (он установлен по умолчанию), а также в сервере приложений Oracle WebLogic, если он включен в режиме отладки.

Некоторые серверы, включая Tomcat, распаковывают WAR-файлы в подкаталог каталога `webapps` или в выбранный вами корневой каталог («контекст») вашего веб-приложения, который вы указали в конфигурационном файле сервера. В этом режиме сервер может разрешить вам заменять лишь отдельные файлы, что особенно полезно при отладке файлов HTML и JSP. Tomcat автоматически перезагружает WAR-файлы при их изменении (если только вы не отключите эту функцию); все, что для этого нужно, — скопировать обновленный WAR-файл поверх старого. В отдельных случаях вам понадобится перезапустить сервер, чтобы изменения вступили в силу.

Безграничный интернет

Мы в самых общих чертах показали, что вы сможете делать с помощью Java и веб-серверов. В этой главе были рассмотрены встроенные средства Java, благодаря которым ваша работа с онлайн-ресурсами будет такой же простой, как с файлами. Ваши приложения будут доступны с любого компьютера Всемирной сети, если вы разместите их на любом подходящем сервере в виде сервлетов. Разрабатывая сервлеты, вы наверняка будете включать в них разнообразные сторонние библиотеки — так же, как это было сделано с файлом `servlet-api.jar`. Надеемся, вы уже примерно представляете себе, как велика современная «экосистема» языка Java.

Количество библиотек и других дополнений к Java увеличивается с каждым годом. И сам язык тоже расширяется и эволюционирует. В следующей главе мы расскажем, как следить за появлением в нем новых возможностей и использовать их для улучшения существующего кода.

ГЛАВА 13

Перспективы Java

Языку Java исполнилось уже более 25 лет. За последнее время он значительно вырос и изменился. Одни изменения были малозаметными и постепенными, другие могут показаться радикальными. Изменился даже сам процесс внесения изменений в язык.

В этой главе мы рассмотрим, с чего начинаются такие изменения и как они в итоге попадают в реальные выпуски Java. Мы продолжим рассказ, начатый в разделе «История Java», с. 47, и перечислим некоторые новинки, предложенные для следующих выпусков. Затем мы вернемся в настоящее и покажем, как можно улучшать ранее написанный код с помощью новых синтаксических конструкций и когда в этом есть смысл. Далеко не все новые возможности Java представляют интерес для каждого разработчика. С другой стороны, почти каждый разработчик найдет для себя что-то интересное в необъятном каталоге возможностей, предоставляемых языком Java и его бесчисленными сторонними библиотеками.

Выпуски Java

На момент издания этой книги версия Java 14 доступна в виде предварительного выпуска. При работе над книгой мы использовали инструментарий разработчика с открытым кодом OpenJDK. Сведения обо всех актуальных выпусках OpenJDK, в том числе планируемых, вы найдете на его сайте: <https://openjdk.org/projects/jdk>. Напомним, что Oracle поддерживает свой официальный JDK, который лучше всего подходит крупным корпоративным клиентам, желающим пользоваться платной поддержкой. За новыми версиями вы можете следить на сайте Oracle (<https://www.oracle.com/java/technologies/java-se-glance.html>). Для всех актуальных версий Oracle JDK там также есть документация о внесенных в них изменениях (release notes): <https://www.oracle.com/java/technologies/javase/jdk-relnotes-index.html>. После Java 9 корпорация Oracle перешла на короткий полугодовой цикл выпуска версий, которые вносят в язык относительно небольшие изменения,

в основном сводящиеся к добавлению отдельных синтаксических инструментов. Короткий цикл означает, что Java регулярно обновляется. Вы можете регулярно загружать новые версии и применять появляющиеся в них инструменты в своем коде. Есть и другой подход — придерживаться какой-то одной стабильной версии с долгосрочной поддержкой, например Java 8 или Java 11. Как мы уже говорили, не все изменения в Java будут вам полезны. Но все равно старайтесь оценивать новые возможности, чтобы понимать, что вас ожидает в ближайшем будущем.

JCP и JSR

Объяснение того, как в Java появляются новые возможности, начнется с нескольких новых сокращений. Общественный процесс JCP (Java Community Process) был создан для того, чтобы привлечь мировое сообщество программистов к формированию путей развития Java. В этом процессе создаются и прорабатываются запросы на спецификацию Java, или JSR (Java Specification Requests). Каждый JSR представляет собой документ, в котором описывается конкретная идея для реализации и разработки группой программистов. Например, в JSR 376 (<https://jcp.org/en/jsr/detail?id=376>) описывается система модулей платформы Java, упрощающая работу с компонентами, необходимыми для создания и развертывания Java-приложений. Вы можете просмотреть все JSR (<https://jcp.org/en/jsr/all>), если вам интересно, что вас ожидает в ближайшем будущем, включая идеи, которые были предложены, но в итоге отклонены или отозваны. Любые JSR, представляющие достаточный интерес, могут занять свое место в виде предварительной реализации функциональности в предстоящей версии Java. Если идея еще не готова для полноценной спецификации, она может быть оформлена в виде предложения по улучшению Java, или JEP (Java Enhancement Proposal) (<https://openjdk.org/jeps/0>). Не все предложения выходят из этой стадии, но как видно, существует достаточно стабильная среда для проверки новых идей и продвижения наиболее перспективных кандидатов для включения в язык.

Если вас интересует, какие возможности войдут в следующий выпуск, посетите сайт сборок Oracle JDK (<https://jdk.java.net>). Здесь вы найдете текущие версии JDK, а также версии, доступные для раннего ознакомления. В сборки раннего доступа (early access) включаются примечания относительно того, что осталось за кадром. Тем не менее при знакомстве с версиями раннего доступа следует серьезно отнестись к первому предупреждению на сайте: «Функциональность раннего доступа (EA, Early Access) может не войти в общедоступный (GA, General Availability) выпуск». Собственно, сами выпуски Java упаковываются в «обобщающие» JSR, такие как JSR 337 для Java 8. Эти обобщающие JSR могут показаться суховатыми, но они содержат официальное описание того, чего следует ожидать от предстоящего выпуска.

Лямбда-выражения

В упомянутом JSR 337 были предложены значительные изменения в Java. Предыдущее издание этой книги завершалось описанием Java 7. Конструкция «try с ресурсами» (наряду со многими другими нововведениями), рассмотренная в главе 11, была потрясающим новшеством. Разработчики в то время уже искали новые возможности, которые обсуждались, но в итоге не были включены в язык. Одним из самых ожидаемых нововведений стали лямбда-выражения (JSR 335). Они позволяют интерпретировать фрагмент кода как полноценный объект. (В контексте лямбда-выражений такие фрагменты кода называются *функциями*.) Если вам не нужно использовать эти функции в других местах кода, то они позволяют писать более компактные программы (и более понятные, когда вы привыкнете к синтаксису). Как и анонимные классы, лямбда-выражения могут обращаться к локальным переменным в той области видимости, в которой они записаны. Они хорошо работают с такими функционально-ориентированными API, как упомянутый в JSR 335 пакет `java.util.stream`. Эти впечатляющие нововведения вошли в Java 8.

Лямбда-функции¹ позволяют рассматривать решение задач в функциональном контексте. Функциональное программирование относится к декларативному стилю программирования. Основное место в нем занимает написание функций (методов со специфическими ограничениями) вместо манипуляций с объектами. Мы не будем углубляться в подробности функционального программирования, но это мощная парадигма, которая будет достойна вашего внимания, когда вы продолжите свое путешествие в мире программирования. Некоторые хорошие книги по функциональному программированию представлены в разделе «За пределами базовых возможностей Java», с. 495, в конце главы.

¹ Джеймс Эллиот (James Elliott), рецензент этой книги и автор книг O'Reilly, предоставил краткую историческую справку: «Термин “лямбда-функции” появился из-за того, что они были изобретены как часть лямбда-исчисления, которое в 1930-х ввел математик Алонзо Черч для получения строгих математических определений вычислительных процессов. Первое появление лямбда-функций в реальном языке программирования произошло почти случайно в 1958 году, когда студенты MIT поняли, как просто реализовать работающую версию языка Lisp, который профессор Джон Маккарти использовал для математического описания анализируемых компьютерных программ. Сейчас Lisp остается вторым по возрасту высокоуровневым языком из применяемых на практике (Fortran на год старше); его исключительная выразительность позволила ему впервые воплотить концепции, нашедшие широкое применение в других языках лишь по прошествии долгого времени, — прежде всего, это уборка мусора и динамическая типизация».

Переработка существующего кода

Лямбда-выражения выглядят довольно заманчиво. Предположим, вы захотите использовать их в своем коде. Что для этого требуется? Отличный вопрос, который относится к любой новой возможности языка. Как мы упоминали, график выпусков Java означает, что вы будете постоянно иметь дело с новыми версиями. Рассмотрим лямбда-выражения с позиций оценки и потенциальной интеграции новых возможностей Java.

Анализ новых возможностей

При рассмотрении любой новой возможности прежде всего надо понять, что она собой представляет. Это может быть как тривиальное изменение синтаксиса, так и нечто сложное, например новый способ компиляции в двоичные файлы. Лямбда-выражения располагаются где-то посередине. Скоро мы рассмотрим очень простое выражение, а затем используем его в коде.

С чего начать знакомство с лямбда-выражениями? Если у вас уже есть опыт программирования на функциональных языках (таких, как Lisp), возможно, вы уже знаете, что такое лямбда-выражения и где они могут применяться. А если этот термин вам еще не встречался, поищите информацию в интернете. Когда какой-либо инструмент существует уже достаточно давно (как лямбда-выражения в Java на момент написания этой книги в начале 2020 года), вы наверняка найдете хорошие руководства. А если инструмент появился совсем недавно или поиски не дали полезных результатов, обратитесь к соответствующему JSR. О лямбда-выражениях также можно узнать в JSR 335 (<https://www.jcp.org/en/jsr/detail?id=335>). Во второй части каждого JSR («Section 2: Request») обычно есть полезная информация. Вот перевод краткого описания функциональности из первого абзаца в разделе 2.7 (перевод):

Мы предлагаем расширить язык Java для поддержки компактных лямбда-выражений (также известных как «замыкания» или «анонимные методы»). Кроме того, язык будет расширен поддержкой так называемых «преобразований SAM», которые позволяют использовать лямбда-выражения там, где ожидается интерфейс с одним абстрактным методом или класс, обеспечивающий совместимость с последующими версиями существующих библиотек.

JSR 335, раздел 2.7

В этих нескольких фразах есть ряд слов, которые помогут вам в поисках справочного материала. Что такое «замыкания» (closures)? Что такое «преобразования SAM» (SAM conversion)? Последнее предложение даже подсказывает, где должны использоваться лямбда-выражения: там, где разрешен конкретный тип интерфейса или класса. Конечно, этого абзаца недостаточно для того, чтобы

в полной мере понять суть лямбда-выражений, но все же он указывает направления для дальнейшего изучения темы.

В оставшейся части JSR содержится больше ссылок на документацию, чем вы сможете прочитать. Возможно, вы найдете ссылки, которые принесут явную пользу, но чаще вам будут попадаться ссылки на вспомогательные материалы, на проектную документацию участников работающей над JSR группы и даже на более ранние черновые версии запроса, по которым видна его эволюция. В документации также должна быть точная информация о той версии Java, в которую включена интересующая вас функциональность (для нашего примера с лямбда-выражениями — Java 8). Даже в сборках раннего доступа должна присутствовать какая-то официальная документация.

Вы можете начать свое знакомство с лямбда-выражениями прямо сейчас. Почитайте сопроводительные документы из JSR. Ознакомьтесь с учебниками по лямбда-выражениям в Java от корпорации Oracle. Поищите информацию в интернете на таких сайтах, как Stack Overflow. Освойте поиск примеров в источниках, которым вы доверяете. Новые версии Java выходят каждые полгода. Научитесь быть в курсе последних событий, и вы не пожалеете о потраченном времени!

Основы лямбда-выражений

Мы надеемся, что вы исследуете эту тему самостоятельно, но все же покажем на примерах компактность и мощь лямбда-выражений. Их базовый синтаксис прост:

(параметры) -> выражение или блок

Часть *параметры* содержит ноль или более именованных (и возможно, типизованных) параметров, которые передаются выражению в правой части новым оператором `->`. Выражение (или блок команд в обычной паре фигурных скобок) может вернуть значение или выполнить некоторый код. Например, популярное лямбда-выражение «инкремент» с одним входным параметром выглядит так:

`(n) -> n + 1`

Заметим, что это лямбда-выражение не изменяет значение параметра `n`, оно просто выполняет вычисление. Этот конкретный пример можно рассматривать как операцию «следующее значение» для целочисленного ряда. Если у вас имеется некоторый контекст, который использует метод `next()` для выполнения некоторой работы, вы можете использовать это лямбда-выражение. Его возможности расширяются, когда тот же контекст нужен для работы с другими типами объектов (например, строками или датами). Какую дату считать «следующей»? Следующий день? Следующий год? С лямбда-выражением можно предоставить специализированную версию «следующего» элемента там, где она потребуется.

Функции можно передать несколько параметров. А можно не передавать ни одного. На практике встречаются все эти варианты, включая популярную сокращенную запись: если параметр ровно один, то круглые скобки в левой части выражения не нужны. Все следующие выражения допустимы:

```
// Один параметр
(n) -> n + 1

n -> n + 1

n -> System.out.println("Working on " + n)

// Без параметров
() -> System.out.println("Done working")

// Несколько параметров
(a, b, c) -> (a + b + c) / 3
```

Возьмем сортировку списков. Если имеется список чисел (в этом примере будет использоваться класс-обертка `Integer`), его сортировка выполняется элементарно:

```
jshell> ArrayList<Integer> numbers = new ArrayList<>();
numbers ==> []

jshell> numbers.add(19)
$5 ==> true

jshell> numbers.add(6)
$6 ==> true

jshell> numbers.add(12)
$7 ==> true

jshell> numbers.add(7)
$8 ==> true

jshell> numbers
numbers ==> [19, 6, 12, 7]

jshell> Collections.sort(numbers)

jshell> numbers
numbers ==> [6, 7, 12, 19]
```

Но что, если числа должны следовать в обратном порядке? Ранее нам пришлось бы написать специальный класс, реализующий интерфейс `Comparator`, или предоставить анонимный внутренний класс:

```
jshell> Collections.sort(numbers, new Comparator<Integer>() {
...>     public int compare(Integer a, Integer b) {
```

```
...>     return b.compareTo(a);  
...>   }  
...> })
```

```
jshell> numbers  
numbers ==> [19, 12, 7, 6]
```

Анонимный внутренний класс работает, но решение получается немного громоздким. Вместо этого можно воспользоваться лямбда-выражением для получения более компактной версии:

```
jshell> Collections.sort(numbers) // Массив отсортирован по возрастанию
```

```
jshell> numbers  
numbers ==> [6, 7, 12, 19]
```

```
jshell> Collections.sort(numbers, (a, b) -> b.compareTo(a))
```

```
jshell> numbers  
numbers ==> [19, 12, 7, 6]
```

Ого! Получается намного элегантнее. Конечно, вы должны понимать, какие аргументы ожидает получить метод `Collections.sort()`, и знать, что интерфейс `Comparator` содержит только один абстрактный метод (то есть является интерфейсом с одним абстрактным методом, или интерфейсом SAM, — вспомните описание из JSR). Но в подходящей среде лямбда-выражения могут быть весьма эффективными.

Новыми инструментами можно воспользоваться для переработки нескольких примеров с генерированием списков. Возьмем фрагмент из раздела «Операции с объектами `File`», с. 403, в котором использовались объекты `java.io.File`. Мы можем отсортировать и вывести имена с использованием объектов `File` и метода `Arrays.asList()` (для получения `Iterable`), с последующим использованием лямбда-выражения с методом `forEach()`:

```
File tmpDir = new File("/tmp" );  
File [] files = tmpDir.listFiles();
```

```
Arrays.sort(files, (a,b) -> a.getName().compareTo(b.getName()))  
Arrays.asList(files).forEach(n -> System.out.println(n.getName()))
```

В этом случае результата можно добиться и без лямбда-выражений, но во многих случаях код с лямбда-выражениями получается более компактным. Конечно, к синтаксису лямбда-выражений нужно привыкнуть, но для этого и нужны практические упражнения!

Ссылки на методы

Этот процесс сортировки сложных объектов по одному из их атрибутов встречается так часто, что в Java API есть вспомогательный метод, создающий нужную

функцию. Статический метод `Comparator.comparing()` помогает написать нечто похожее на лямбда-выражение, использующее `compareTo()` в предыдущем разделе. В нем используются *ссылки на методы* — упрощенные разновидности лямбда-выражений, в которых вызываются существующие методы из других классов.

У ссылок на методы есть много подробностей и сценариев использования, в которые мы не будем углубляться, но базовый синтаксис и использование достаточно просты. Имя класса отделяется от имени метода разделителем из двух двоеточий (`::`). Метод `Comparator.comparing()` ожидает получить ссылку на метод, который может использоваться с сортируемыми объектами (например, можно вызывать соответствующие методы). При сортировке объектов `File` можно использовать любые `get`-методы, возвращающие информацию, которая может использоваться при сортировке, например имя или размер файла:

```
Arrays.sort(files, Comparator.comparing(File::getName));
```

Все вполне чисто! И мы точно видим, что намерен сделать программист: отсортировать группу файлов, сравнивая их имена. Это именно то, что мы делали с помощью лямбда-выражений в предыдущем разделе. Помните: дело не в том, что ссылки на методы лучше, — их всегда можно заменить лямбда-выражениями; во многих случаях ссылки на методы (как и лямбда-выражения) могут сделать код нагляднее, если вы привыкнете к новому синтаксису¹.

Лямбда-выражения с событиями

Мы видели несколько других примеров кода с аналогичными ограничениями среды. Вспомните многочисленные обработчики событий в главе 10. Некоторые слушатели точно соответствовали схеме одного абстрактного метода — как интерфейс `ActionListener`, используемый `JButton` или `JMenuItem`. Там, где это уместно, мы можем использовать лямбда-выражения, чтобы упростить код обработки событий. Часто в программе используются простые временные обработчики для проверки базовой возможности щелчка на кнопке:

```
JButton okButton = new JButton("OK");
okButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        System.out.println("OK pressed!");
    }
});
```

При помощи лямбда-выражений этот код можно заметно сократить. В результате у вас получится примерно такой тестовый код для нескольких кнопок:

¹ Посмотрите на сайте Stack Overflow мнение Брайана Гетца (Brian Goetz) по этому вопросу (<https://stackoverflow.com/questions/24487805/lambda-expression-vs-method-reference>).

```
JButton okButton = new JButton("OK");
okButton.addActionListener(ae -> System.out.println("OK pressed"));
JButton noButton = new JButton("Cancel");
noButton.addActionListener(ae -> System.out.println("Cancel pressed"));
```

Превосходно! Лямбда-выражения могут предоставить удобное решение для ситуаций, в которых требуется небольшой объем динамического кода. Конечно, не все обработчики событий подходят для подобных преобразований. Тем не менее во многих случаях они оказываются уместными, а компактная запись сделает ваш код более простым для понимания.

Замена Runnable

Другой популярный интерфейс, соответствующий этой модели, — интерфейс `Runnable`, с которым мы работали в главах 9 и 10. Мы рассматривали примеры использования внутренних классов и анонимных внутренних классов для создания новых объектов `Thread`. Метод `SwingUtilities.invokeLater()` также получает экземпляр `Runnable` в аргументе. Лямбда-выражения могут использоваться и в этих случаях. Вспомните пример `ProgressPretender` из раздела «`SwingUtilities` и обновление компонентов», с. 380. Мы уже находились внутри метода `run()` класса, реализующего интерфейс `Runnable`, когда потребовалось создать второй, анонимный экземпляр `Runnable` для обновления надписи:

```
public void run() {
    while (progress <= 100) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                label.setText(progress + "%");
            }
        });
    }
    // ...
}
```

Но теперь можно воспользоваться лямбда-выражением для того, чтобы не отвлекаться от реальной работы, выполняемой потоком:

```
public void run() {
    while (progress <= 100) {
        SwingUtilities.invokeLater(() -> label.setText(progress + "%"));
    }
    // ...
}
```

И снова код получается куда более компактным — и надеемся, более удобочитаемым. Это изменение не является обязательным, оно не ускоряет работу приложения, но если вы (и участники вашей рабочей группы) понимаете лямбда-выражения, то такой код может упростить сопровождение и оставить больше времени для работы над другими задачами.

За пределами базовых возможностей Java

Важно отметить, что многие части Java используют JSR за пределами базовых возможностей языка. Например, в JSR 369 рассматривается спецификация сервлетов Java 4.0. Как упоминалось в разделе «Сервлеты», с. 467, для компиляции и запуска примеров сервлетов понадобится отдельный файл `servlet-api.jar`. Просмотрев описание в JSR 469, мы видим, что спецификация 4.0 создавалась для поддержки функциональности, входящей в HTTP/2. Углубленное изучение этой функциональности показывает, что одним из самых долгожданных нововведений стала поддержка *отправки по инициативе сервера (server push)* — так называется способность сервера ускорять доставку сложных страниц путем «активной отправки» некоторых файлов или ресурсов еще до момента их непосредственного использования клиентом.

В протоколе HTTP/1.1 страница HTML доставляется браузеру при посещении сайта. В свою очередь, эта страница приказывает браузеру запросить другие ресурсы: файлы JavaScript, стилевые таблицы, графику и т. д. Каждый из этих ресурсов требует отдельного запроса. Кэширование отчасти ускоряет этот процесс, но при первом посещении нового сайта кэш еще пуст, так что время загрузки может быть весьма значительным. HTTP/2 позволяет серверу отправлять ресурсы заранее, эффективнее используя канал связи. Эта оптимизация ускоряет получение страницы, даже если она содержит данные, которые не кэшируются (или не могут кэшироваться).

В настоящее время переход на HTTP/2 сам по себе является довольно сложным делом. Далеко не каждый сайт использует HTTP/2, и не каждый браузер его поддерживает (или поддерживает лишь частично). Здесь эта тема рассматриваться не будет, но вам лучше изучить ее самостоятельно, если веб-программирование относится к вашей повседневной работе. В любом случае не забывайте, что на сайте JCP можно узнавать, что ожидается в Java в ближайшее время — как в самом языке, так и в его более широкой экосистеме.

Заклучение и следующие шаги

В этой главе мы только в самых общих чертах затронули лямбда-выражения и соответствующие части Java API, включая ссылки на методы и API потоков данных. К сожалению, как и во многих других интересных темах, затронутых в книге, мы вынуждены оставить дальнейшие исследования вам. К счастью, Java 8 существует уже много, много лет¹, поэтому сетевых ресурсов, посвященных этим возможностям, предостаточно. Лямбда-выражения и функцио-

¹ Интересно, что Java 8 по данным различных отраслевых опросов 2019 года остается одной из самых распространенных версий языка.

нальное программирование в Java подробно рассматриваются в книге Ричарда Уорбертона (Richard Warburton) «Java 8 Lambdas», вышедшей в издательстве O'Reilly. В мире сервлетов существует более новая спецификация версии 4, но в интернете все равно можно найти немало замечательных ресурсов, в которых рассматривается как эта спецификация, так и HTTP/2.

Итак... у вас получилось! Сказать, что в книге изложен большой объем материала, значит не сказать ничего. Надеемся, что книга заложила хорошую основу для более глубокого изучения всех представленных тем и нетривиальных возможностей Java. Выберите область, которая вас интересует, и беритесь за дело. Если вас интересует Java в целом, попробуйте связать воедино все части этой книги. Например, попробуйте написать сервлет для ответов на запросы, сходный с клиентом `DateAtHost` из раздела «Клиент `DateAtHost`», с. 438. Попробуйте использовать регулярные выражения для разбора протокола нашей игры с бросанием яблок. Или разработайте более сложный протокол, чтобы передавать по сети блоки двоичных данных вместо простых строк. Чтобы тренироваться в написании более сложных программ, перепишите некоторые внутренние и анонимные классы в игре в виде автономных классов или воспользуйтесь лямбда-выражениями.

Если вы хотите изучить другие библиотеки и пакеты Java, не расставаясь с примерами, над которыми вы уже работали, займитесь изучением `Java2D API` и нарисуйте более симпатичные яблоки и деревья. Исследуйте формат `JSON` и попробуйте переписать сервлеты `ShowParameters` и `ShowSession`, чтобы они возвращали блок разметки `JSON` вместо страницы `HTML`. Попробуйте использовать другие объекты коллекций, например `TreeMap` или `Stack`.

А если вы готовы заглянуть еще глубже, ознакомьтесь с тем, как Java работает за пределами десктопных систем, и опробуйте разработку для `Android`. Или присмотритесь к очень большим сетевым средам и к `Jakarta Enterprise Edition` от `Eclipse Foundation`. Может быть, ваша работа связана с большими данными? `Apache Foundation` ведет такие проекты, как `Hadoop` и `Spark`. У Java есть свои критики, и все же Java остается живой и энергичной частью мира профессиональной разработки.

Итак, вас ждут все эти возможности, а основная часть книги подходит к концу. В глоссарии приведена краткая сводка многих полезных терминов, которые могут вам пригодиться. В приложении рассказано об установке среды разработки `IntelliJ IDEA`, а также об импортировании и запуске примеров кода. Надеемся, что книга вам понравилась. Это пятое издание под названием «Программируем на Java» («`Learning Java`») на самом деле является седьмым изданием нашей серии, которая началась более 20 лет назад с книги «`Exploring Java`». На этом долгом и интересном пути мы следили за тем, как Java развивается с течением времени, и мы благодарны тем читателям, которые были с нами все эти годы. Как всегда, мы ждем ваших отзывов, которые помогут улучшить эту книгу в будущем. Вы готовы отметить следующее десятилетие Java? Тогда мы с вами!

Примеры кода и IntelliJ IDEA

Это приложение поможет вам начать работу с примерами кода, приведенными в книге. Некоторые из этих операций упоминались в главе 2, но здесь мы приводим более подробное описание, уделяя внимание работе с кодом в бесплатном издании (Community Edition) среды разработки IntelliJ IDEA от компании JetBrains.

Отметим, что IntelliJ IDEA — не единственная Java-совместимая интегрированная среда разработки (IDE). Это даже не единственная бесплатная IDE. Например, одна из альтернатив — это Microsoft VS Code (<https://code.visualstudio.com/download>), которая быстро настраивается для поддержки Java. Также существует IDE Eclipse (<https://www.eclipse.org/downloads/packages>), сопровождением которой занимается IBM. А начинающему Java-программисту, которому нужен инструмент, упрощающий первые шаги в мире программирования, имеет смысл присмотреться к IDE BlueJ (<https://www.bluej.org>), разработанной Королевским колледжем Лондона.

Где загрузить примеры кода

Независимо от того, какой IDE вы пользуетесь (и пользуетесь ли вообще), мы рекомендуем загрузить примеры кода с GitHub. Хотя мы часто приводим полные листинги при обсуждении конкретных тем, для краткости и наглядности мы часто исключали из них второстепенные элементы (команды `import` или `package`, завершающие строки классов и т. д.). Примеры на GitHub содержат полный код, чтобы вы могли открыть их в текстовом редакторе или в IDE, просмотреть, скомпилировать и запустить. Это необходимо, чтобы хорошо понять написанное в книге.

Вы также можете зайти на GitHub в браузере, чтобы просмотреть отдельные примеры без загрузки. Просто откройте репозиторий `learnjava5e` (<https://github.com/I0y/learnjava5e>). Если эта ссылка не работает, откройте <https://github.com> и проведите поиск по строке «`learnjava5e`». Возможно, вам вообще стоит осмотреться

на GitHub, так как этот портал стал основной рабочей площадкой для сообщества разработчиков ПО с открытым кодом и даже для корпоративных проектов. Вы можете просмотреть историю репозитория и отчеты об ошибках, а также обсудить вопросы, связанные с кодом.

Имя сайта происходит от названия программы `git` — системы управления исходным кодом, которую программисты используют для контроля изменений при работе над проектами в группах. Если на вашей платформе нет этой программы, загрузите ее по адресу <https://git-scm.com/downloads>. GitHub также поддерживает сайт try.github.io, который поможет вам освоить возможности `git`. После того как программа `git` будет установлена, вы можете клонировать проект в папку на вашем компьютере. Вы можете работать с клонированной копией или хранить ее как эталонную версию примеров кода. Если мы опубликуем какие-либо обновления или исправления, вы сможете легко синхронизировать клонированную папку.

Также можно загрузить весь набор примеров из главной ветви проекта в виде ZIP-архива (<https://oreil.ly/y4nNh>). Если эта ссылка не работает, поищите кнопку `Code` на главной странице репозитория `learnjava5e`. После загрузки файла `learnjava5e-master.zip` распакуйте его в ту папку, в которой вам будет удобно работать с примерами. Структура папок должна выглядеть примерно так, как показано на рис. П.1.

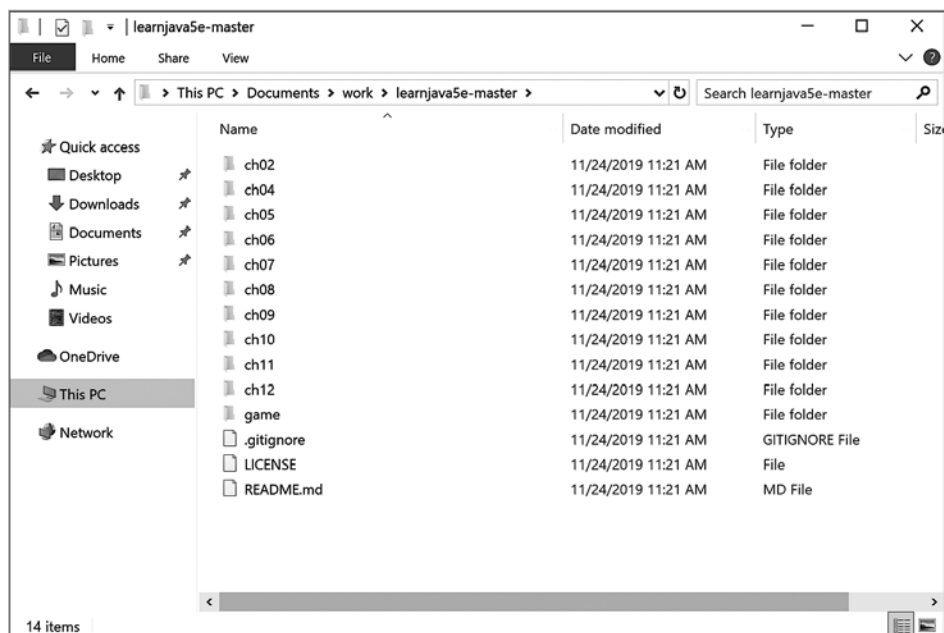


Рис. П.1. Структура папок с примерами кода

В следующих разделах мы расскажем, как настроить и запустить IntelliJ IDEA, после чего займемся импортированием примеров кода.

Установка IntelliJ IDEA

Чтобы начать работу, зайдите на сайт JetBrains и загрузите копию издания Community Edition (<https://www.jetbrains.com/idea/download>). Нужную версию можно выбрать в разделе «Other versions» (мы использовали версию 2019.2.4). Обычно сайт успешно определяет вашу платформу, но все же проследите за тем, чтобы был загружен дистрибутив, подходящий для вашей операционной системы (или для той системы, в которой вы собираетесь установить IntelliJ IDEA, если у вас несколько компьютеров).

На сайте JetBrains есть удобное руководство по установке (<https://www.jetbrains.com/help/idea/installation-guide.html>), но на всякий случай мы укажем основные действия для каждой платформы.

Установка в Linux

В системе Linux JetBrains рекомендует установить IntelliJ IDEA в каталоге `/opt`. Конечно, при желании вы можете установить ее и в другом месте. Как и в случае с OpenJDK (см. «Установка OpenJDK в Linux», с. 55), сначала распакуйте файл `tar.gz` в выбранный каталог:

```
~ $ cd Downloads
```

```
~/Downloads $ sudo tar xf ideaIC-2019.2.4.tar.gz -C /opt
```

Чтобы запустить инсталлятор, найдите файл сценария `idea.sh` в каталоге `bin` внутри того каталога, куда распакован загруженный файл. Вам придется принять лицензионное соглашение и ответить на несколько исходных вопросов (например, выбрать цветовую схему и плагины, которые могут понадобиться). После ответа на эти вопросы (которые задаются всего один раз) должен появиться экран приветствия, показанный на рис. П.2.

Основные действия по импортированию примеров описаны в разделе «Импортирование примеров кода», с. 502.

Установка в macOS

Для macOS загрузите файл `.dmg`. Сделайте двойной щелчок на этом файле, чтобы смонтировать его, а затем перетащите файл приложения IntelliJ IDEA в папку



Рис. П.2. Экран приветствия IntelliJ IDEA в Linux

Applications, как бы вы это сделали для любой другой автономной установки в macOS. После того как файл будет скопирован, запустите его и ответьте на вопросы, связанные с лицензией и персональными предпочтениями. Экран должен выглядеть примерно так, как на рис. П.3, хотя скорее всего, вы не увидите списка ранее открывавшихся проектов слева. Если вы впоследствии закроете в IntelliJ IDEA все свои активные окна, то снова появится экран приветствия, а список слева будет заполнен вашими проектами.

Основные действия по импортированию примеров описаны в разделе «Импортирование примеров кода», с. 502.

Установка в Windows

Для Windows загрузите инсталлятор .exe и запустите его. Начнется процедура установки IntelliJ IDEA. Вам будет предложено выбрать, в какую папку вы хотите установить приложение, какие дополнительные компоненты выбрать для установки и т. д.

Когда процесс установки завершится, запустите IntelliJ IDEA. Как и на других платформах, вам придется ответить на несколько вопросов и подтвердить свое согласие на условия лицензии. В итоге появится все тот же экран приветствия, показанный на рис. П.4.

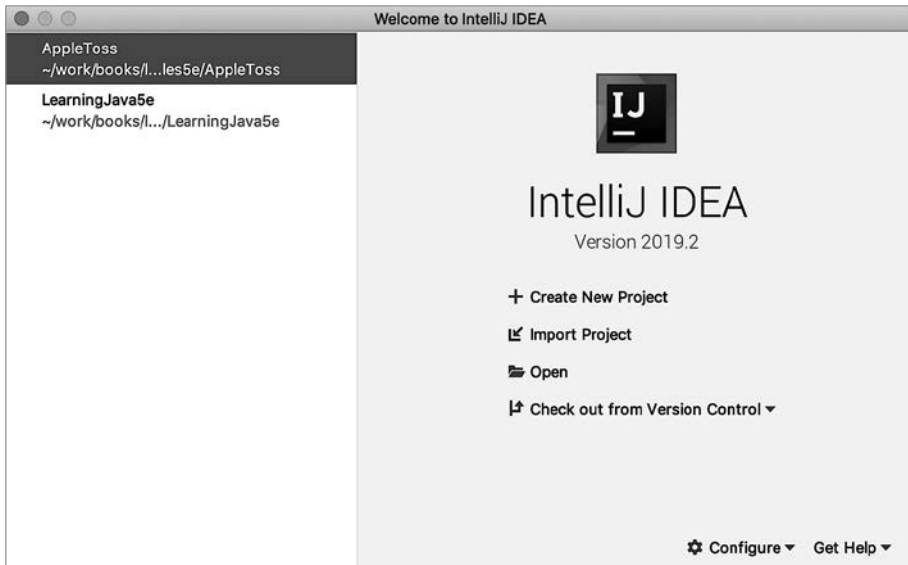


Рис. П.3. Экран приветствия IntelliJ IDEA в macOS (со списком предыдущих проектов)



Рис. П.4. Экран приветствия IntelliJ IDEA в Windows

Теперь рассмотрим процедуру импортирования примеров исходного кода в IntelliJ IDEA, необходимую для последующей удобной работы с ними.

Импортирование примеров кода

Прежде чем рассматривать процесс импортирования в IntelliJ IDEA, желательно переименовать папку, в которую вы загрузили примеры кода с GitHub. Если вы загрузили архив .zip, то после его распаковки примеры кода окажутся в папке с именем `learnjava5e-master`. В этом имени нет ничего плохого, но если вы хотите заменить его более коротким или наглядным, сделайте это сразу, перед началом импортирования. Мы переименовали папку в `LearningJava`.

На экране приветствия IntelliJ IDEA выберите пункт меню **Import Project**. (Если вы уже работали в IntelliJ IDEA и не видите экрана приветствия, выберите команду **File** ► **New** ► **Project from Existing Sources...**) Перейдите в папку с примерами кода (рис. П.5). Проследите за тем, чтобы выбрана была папка верхнего уровня, а не одна из папок отдельных проектов.

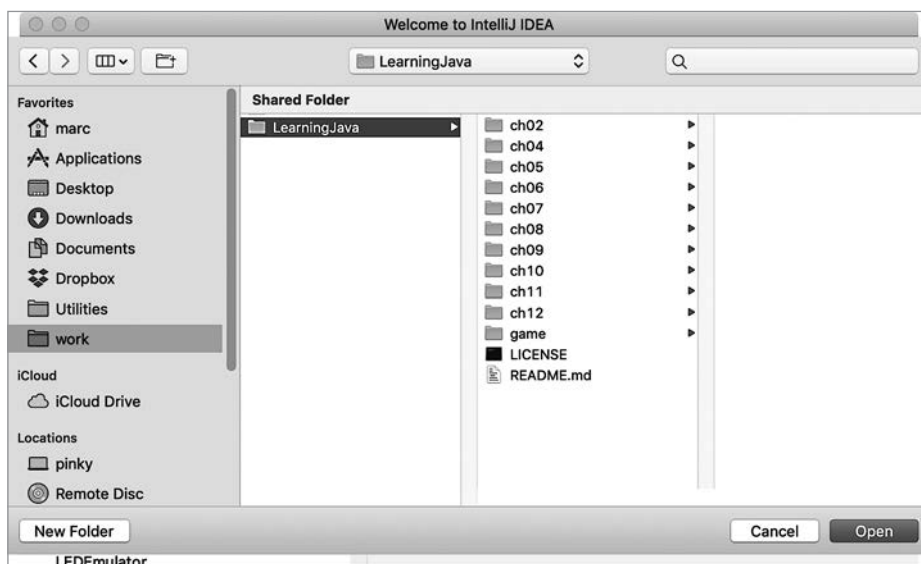


Рис. П.5. Импортирование папки с примерами кода

После открытия папки примеров вам будет предложено просмотреть найденные библиотеки (рис. П.6). В нашем случае таких библиотек нет, поэтому нажмите кнопку **Next**.

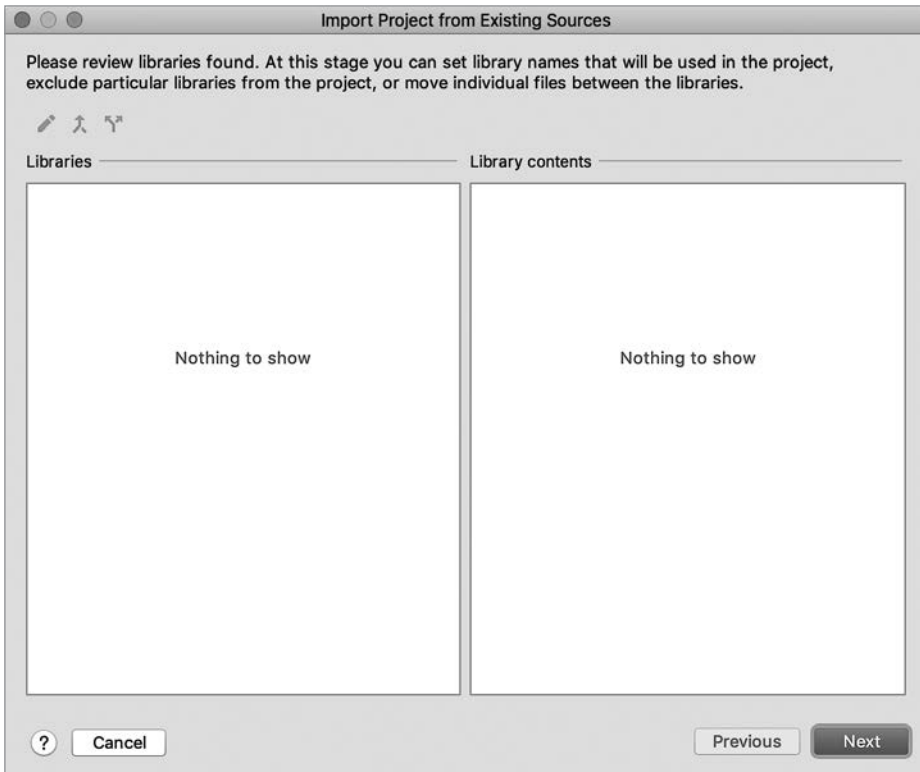


Рис. П.6. Диалоговое окно просмотра библиотек

При желании на следующем экране можно изменить имя проекта, но проследите за тем, чтобы оставалась выбранной папка с примерами кода верхнего уровня. Нас устраивает имя `LearningJava`, поэтому мы оставили оба поля без изменений (рис. П.7).

Исходные файлы должны быть успешно обнаружены. На следующем экране оставьте флажок установленным (рис. П.8) и нажмите кнопку `Next`.

В версии 2019.2.4 вам будет предложено во второй раз просмотреть найденные библиотеки (рис. П.6). В наших простых примерах таких библиотек не будет, поэтому нажмите кнопку `Next`. (Добавление библиотеки сервлетов, необходимой для наших примеров, рассматривается в разделе «Загрузка кода веб-приложений», с. 509.)

В наших примерах поддержка модулей, появившаяся в Java 9, не используется, поэтому на следующем экране оставьте флажок установленным (рис. П.9) и нажмите кнопку `Next`.

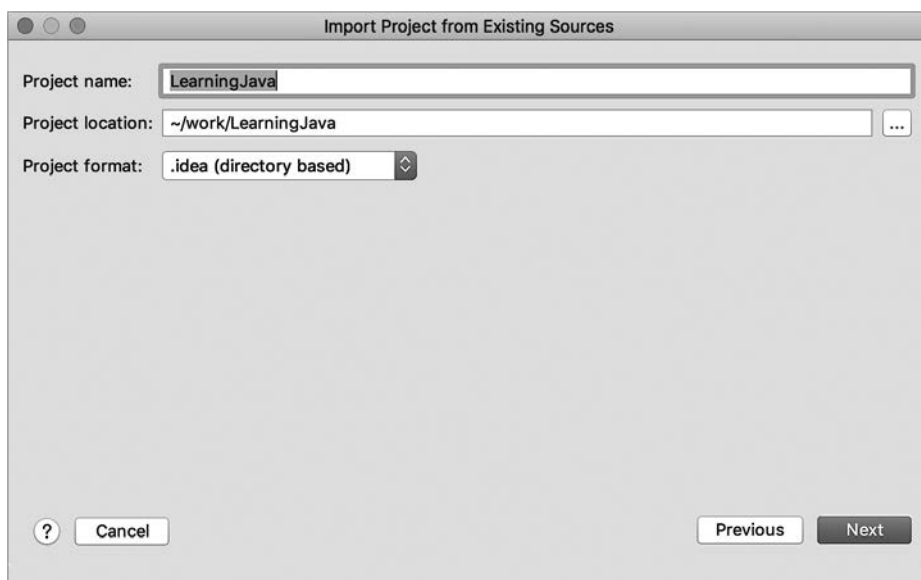


Рис. П.7. Диалоговое окно с именем и местонахождением проекта

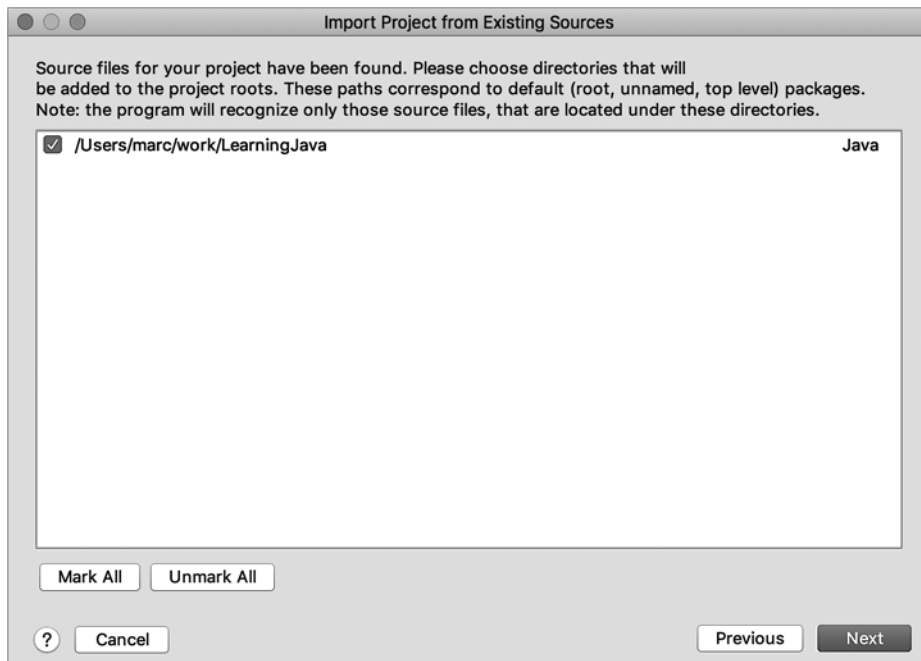


Рис. П.8. Диалоговое окно с исходной папкой

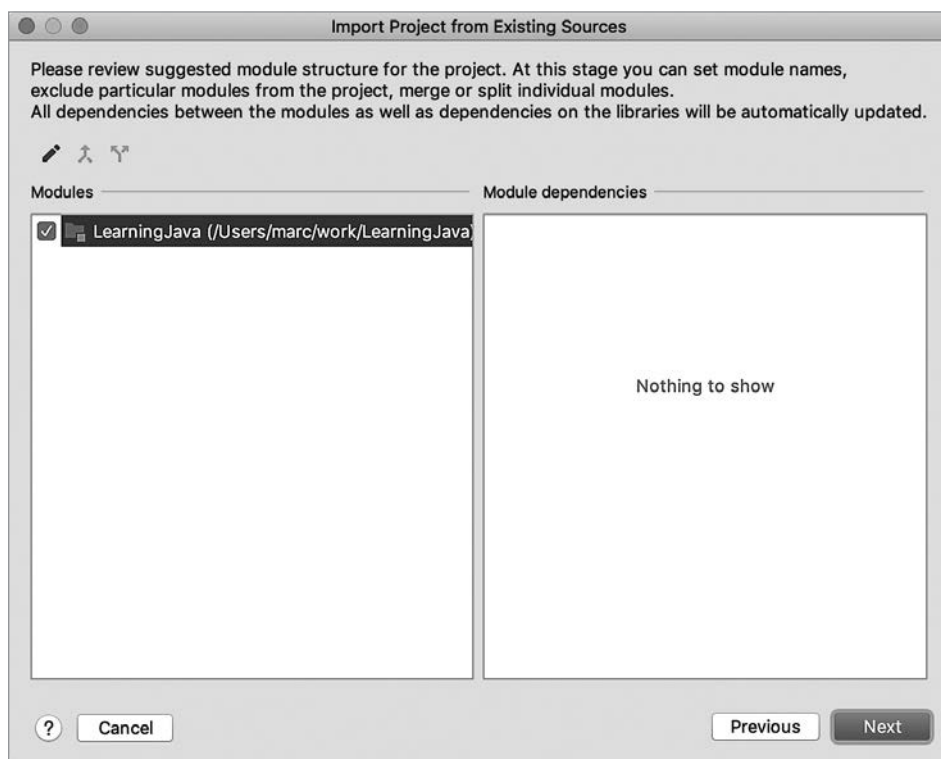


Рис. П.9. Диалоговое окно модулей

Вам будет предложено выбрать SDK (Software Development Kit — в нашем случае это эквивалент версии Java). Мы выбрали версию с долгосрочной поддержкой (11), как видно из рис. П.10, но вы можете выбрать и другую версию (11 или выше), если она у вас установлена. Если вы забыли, как загрузить или установить Java SDK, обращайтесь к разделу «Установка JDK», с. 54.

Нажмите кнопку **Finish**. Проект должен быть готов к работе, как показано на рис. П.11.

Запуск примеров кода

Как упоминалось в главах 2 и 3, вы можете использовать терминал или командную строку для того, чтобы компилировать примеры командой `javac`, а затем запускать их командой `java`. Но поскольку среда IntelliJ IDEA настроена и готова к работе, посмотрим, как запустить эти примеры в IDE.

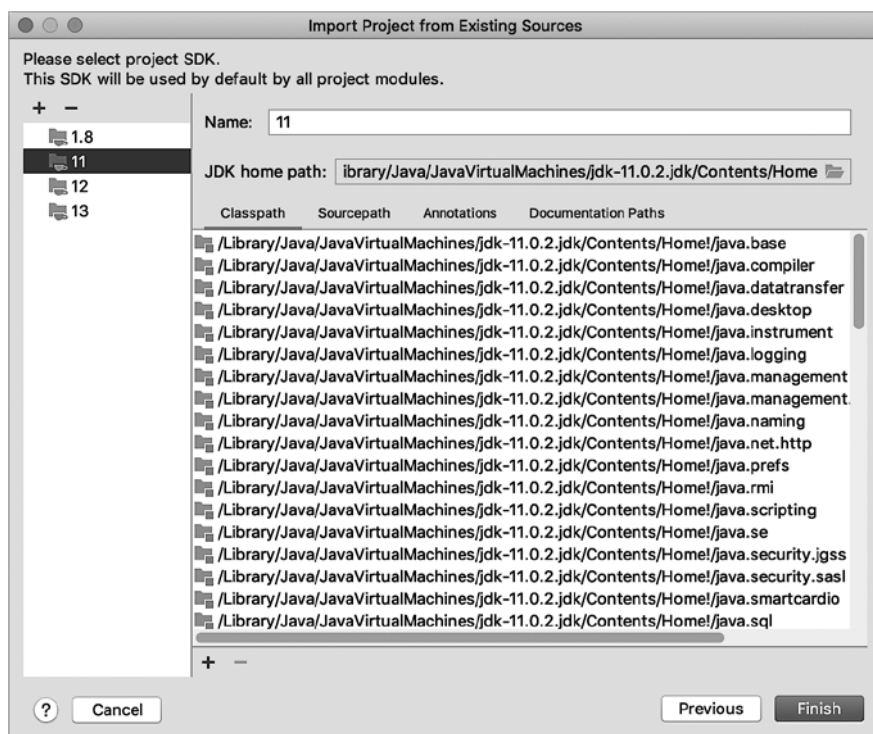


Рис. П.10. Диалоговое окно выбора SDK

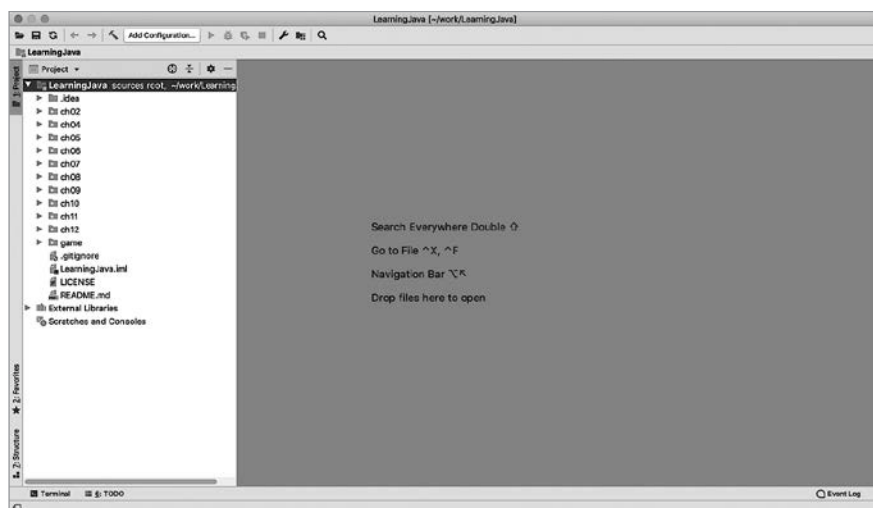


Рис. П.11. IntelliJ IDEA готова к работе!

Перейдите в папку `ch02` проекта и сделайте двойной щелчок на элементе `HelloJava`. В IDE должна появиться вкладка с исходным кодом из файла `HelloJava.java` (рис. П.12).

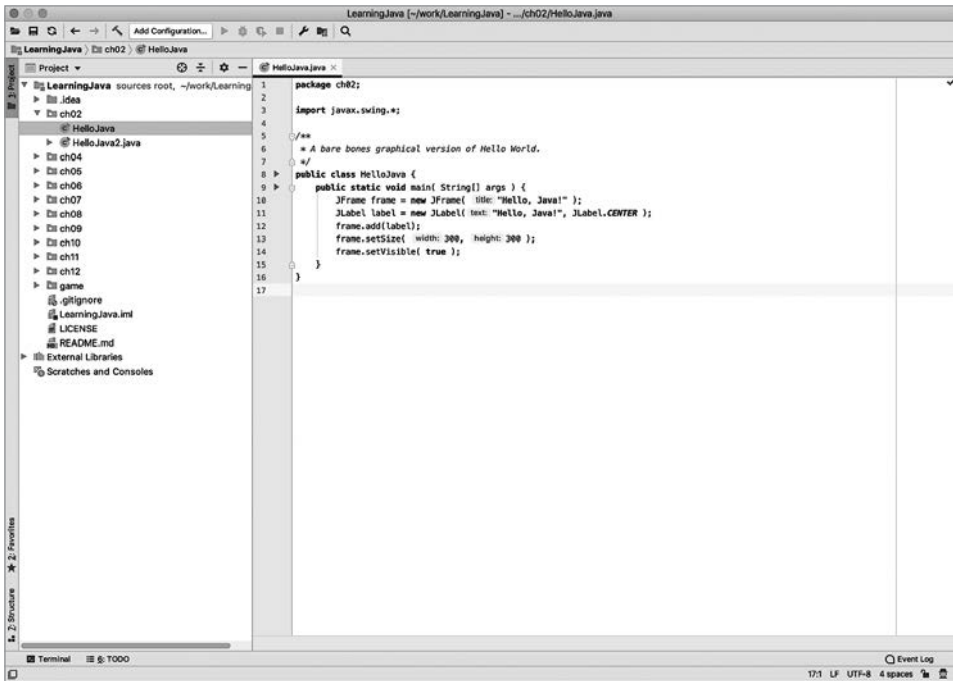


Рис. П.12. Исходный код класса `HelloJava`

Конечно, файл можно отредактировать, но пока оставим его в исходном виде. На панели структуры проекта в левой части найдите элемент `HelloJava`, щелкните правой кнопкой мыши и выберите команду `Run HelloJava.main()` в середине открывшегося контекстного меню (рис. П.13).

После того как вы запустите класс, IntelliJ IDEA обычно закрепляет этот класс по умолчанию за кнопкой выполнения на панели инструментов. С помощью этой кнопки вы сможете быстрее запустить то же приложение — это идеальный вариант, когда вы тестируете новый класс, вносите изменения и тестируете снова. Но при переходе к новому классу вам придется вернуться и запустить новый класс из контекстного меню. В этот момент новый класс будет назначен по умолчанию для кнопки выполнения.

На экране должно появиться уже знакомое (хотя и простое) окно, изображенное на рис. П.14.

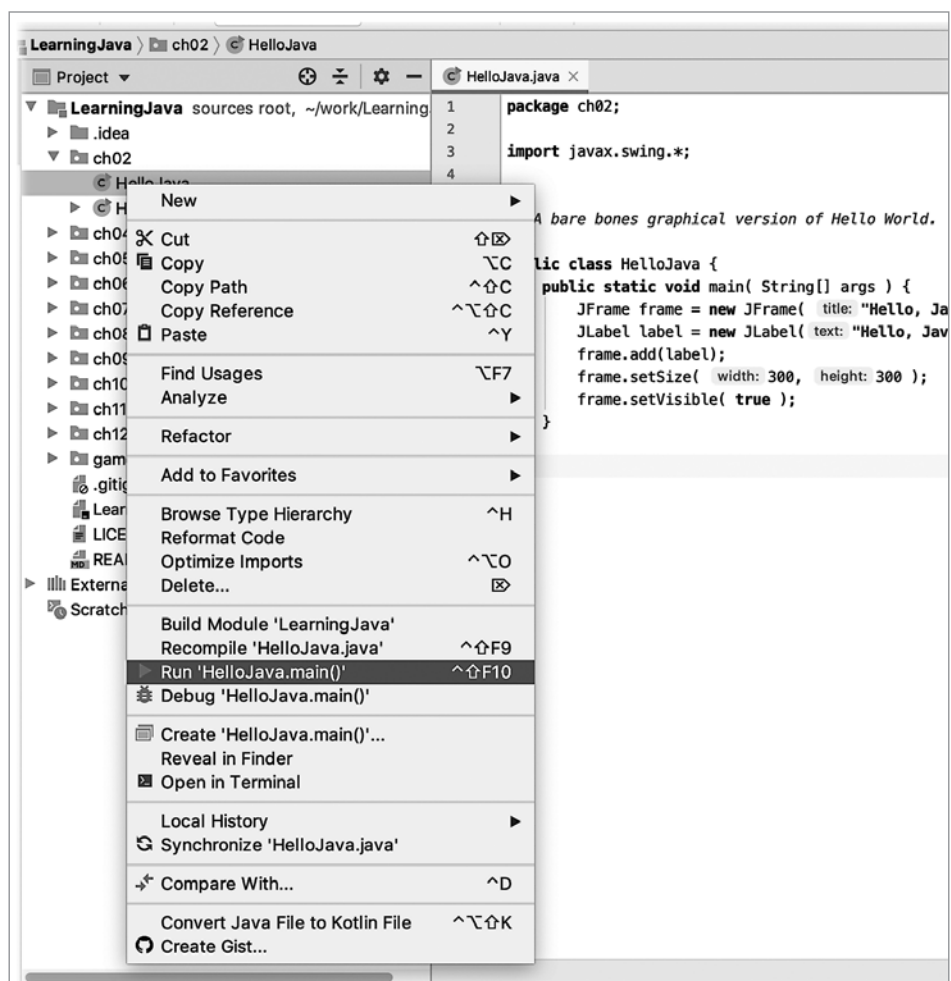


Рис. П.13. Запуск класса из контекстного меню

Поздравляем! Среда разработки IntelliJ IDEA настроена, и все готово к тому, чтобы вы сделали первые шаги в удивительном мире программирования на Java. Если вас не интересует тема веб-программирования на Java, папку ch12 можно исключить. Если же вы планируете опробовать примеры этой главы, — а мы рекомендуем вам это сделать, — продолжайте читать. Вы узнаете, как добавить необходимую библиотеку.

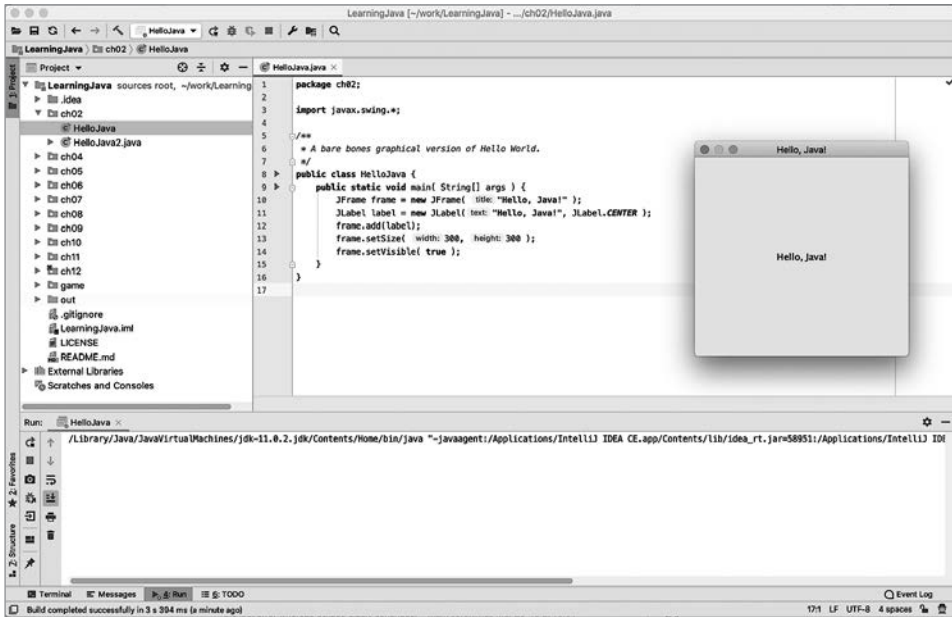


Рис. П.14. Успешный запуск приложения HelloJava

Загрузка кода веб-приложений

Вернитесь на сайт GitHub в браузере и найдите второй репозиторий (<https://github.com/I0y/learnjava5e-web>). (И снова, если ссылка не работает, откройте <https://github.com> и проведите поиск по строке «learnjava5e-web».) Этот репозиторий намного меньше первого, но по своей структуре он похож на основные примеры. Мы специально выделили его, чтобы вы могли сосредоточиться на основных примерах без подключения дополнительных библиотек.

Используйте `git` в терминале, как и прежде, или загрузите ZIP-архив. Если вы загрузили ZIP-архив, распакуйте его. Мы переименовали папку верхнего уровня и присвоили ей имя `LearningJavaWeb`.

Теперь выберите команду `File ▸ New ▸ Project from Existing Sources...` и перейдите в папку с примерами. Проследите за тем, чтобы выбрана была папка верхнего уровня, а не папка `ch12`. Теперь в IDEA должен появиться второй проект, но для сервлетов необходимо сделать еще один шаг.

Работа с сервлетами

В главе 12 рассматривается использование Java в мире веб-программирования. В этой сфере можно делать очень многое, не пользуясь ничем, кроме API, доступных в JDK. Но если вы хотите писать сервлеты, то надо загрузить контейнер и сообщить IntelliJ IDEA, где следует искать библиотеку сервлетов.

Как упоминалось в разделе «Развертывание сервлета HelloClient», с. 484, мы рекомендуем загрузить и установить контейнер Apache Tomcat. Вы можете загрузить новейшую версию и поискать полезную документацию на сайте Apache Tomcat. Здесь также можно загрузить версию 9: <https://oreil.ly/HWY7l>. Выберите .zip или .tar.gz по своему усмотрению. Распакуйте архив в любую папку, где вам будет удобно найти его позднее. Версия 9 потребуется вам, если вы захотите исследовать возможность отправки данных по инициативе сервера, упомянутую в разделе «За пределами базовых возможностей Java», с. 495. Вы найдете информацию о том, какие версии Tomcat поддерживают те или иные версии API сервлетов, на странице «Which Version?» сайта Tomcat (<http://tomcat.apache.org/whichversion.html>).

В IntelliJ IDEA откройте окно Project Structure. Щелкните правой кнопкой мыши на проекте (верхний элемент LearningJavaWeb в нашем случае) и выберите в контекстном меню команду Open Modules Settings (или выберите в меню команду File ► Project Structure...). На экране должно появиться окно, показанное на рис. П.15. Выберите в левом меню пункт Libraries.

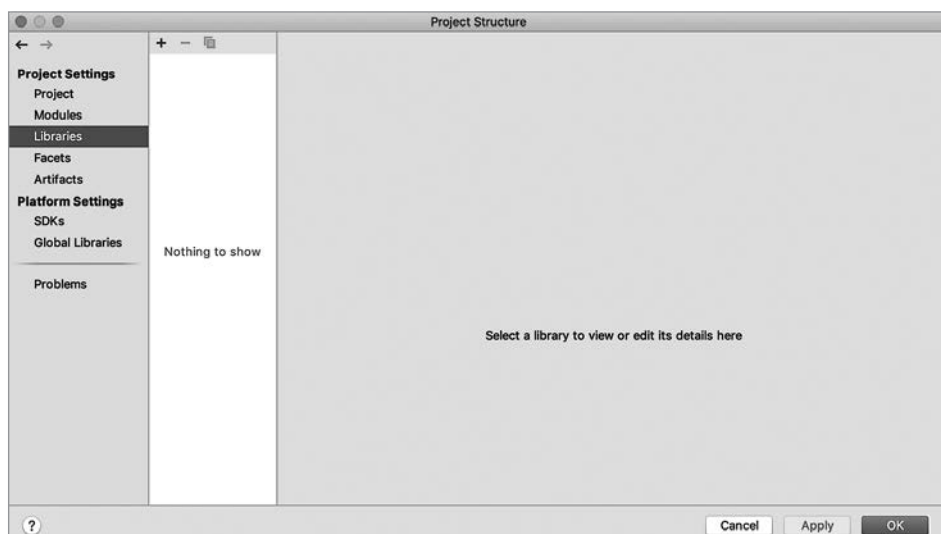


Рис. П.15. Настройки библиотек для проектов

Щелкните на значке + в левом верхнем углу среднего столбца, выберите тип добавляемой библиотеки **Java**. Теперь необходимо перейти в тот каталог, куда вы загрузили и распаковали Tomcat. Нам нужен файл `servlet-api.jar` из библиотеки `lib`, как показано на рис. П.16.

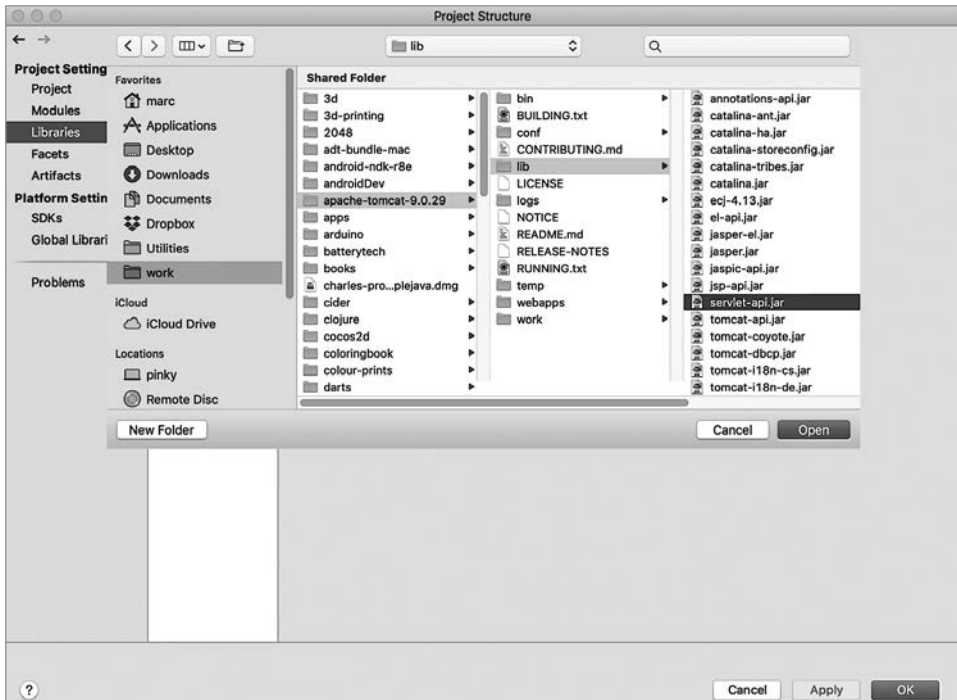


Рис. П.16. Папка `lib` в Tomcat

Выделите файл и щелкните на кнопке **Open**, а затем на кнопке **OK**. Откроется следующее диалоговое окно с информацией о добавлении библиотеки к модулю `LearningJava`. В итоге раздел **Libraries** должен выглядеть так, как показано на рис. П.17. Щелкните на кнопке **OK**.

Чтобы убедиться в том, что библиотека сервлетов установлена правильно, соберите проект командой меню **Build** ▶ **Build Project**. После небольшой паузы IntelliJ IDEA сообщит, что сборка прошла успешно. Вы все еще должны выполнить развертывание, как описано в разделе «Развертывание сервлета `HelloClient`», с. 484, но и сейчас вы сможете пользоваться всеми замечательными возможностями IDE (например, автозавершением) в своих примерах сервлетов.

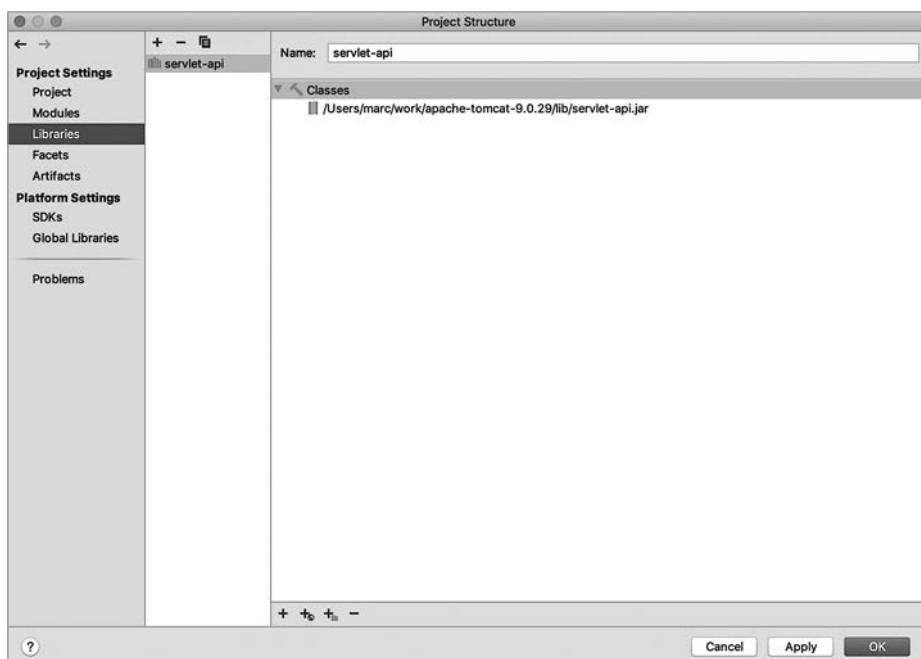


Рис. П.17. Правильно настроенная библиотека сервлетов

Если вы основательно занимаетесь веб-программированием (или собираетесь им основательно заняться), возможно, стоит присмотреться к платному изданию IntelliJ IDEA Ultimate Edition. В него включены некоторые замечательные возможности работы с сервлетами и сопутствующими веб-технологиями. О возможностях Ultimate Edition рассказано на сайте JetBrains, в справке о работе с веб-приложениями (<https://www.jetbrains.com/help/idea/developing-web-applications.html>).

Глоссарий

<object>

Тег HTML, используемый для встраивания мультимедийных данных и приложений в браузеры.

abstract

Ключевое слово **abstract** используется для объявления абстрактных методов и классов. Абстрактный метод не имеет определенной реализации; он, как обычно, объявляется с аргументами и возвращаемым типом, но тело, заключенное в фигурные скобки, заменяется символом ; (точка с запятой). Реализация абстрактного метода обеспечивается субклассом того класса, в котором этот метод был определен. Если класс содержит абстрактный метод, то этот класс тоже становится абстрактным. Попытка создать экземпляр абстрактного класса приведет к ошибке компиляции.

Ant

Традиционная утилита сборки приложений Java, основанная на XML-сценариях. Ant может компилировать исходный код Java, составлять пакеты, готовить приложения к распространению, генерировать документацию и выполнять другие операции, руководствуясь указанными в сценариях «целями».

API (Application Programming Interface)

Интерфейс программирования приложений, состоящий из классов, методов и переменных, которые программисты используют для работы с программными и аппаратными компонентами. Основные API языка Java содержатся в пакетах `java.lang`, `java.util`, `java.io`, `java.text`, `java.net` и во многих других.

API коллекций (Collections API)

Классы и интерфейсы из фундаментального пакета `java.util`, предназначенные для работы со структурированными коллекциями и картами (ассоциа-

тивными массивами) элементов. В этот API входят традиционные классы (такие, как `Vector` и `Hashtable`), а также многие более новые элементы: `List`, `Map`, `Queue` и т. д.

API конфигураций (Preferences API)

API для хранения небольших объемов конфигурационных данных (относящихся к отдельным пользователям или ко всей системе в целом) в периоды между запусками виртуальной машины Java. API конфигураций работает подобно небольшой базе данных или реестру Windows.

API регулярных выражений (regular expressions API)

Фундаментальный пакет `java.util.regex`, предназначенный для работы с регулярными выражениями. Пакет `regex` может использоваться для поиска и замены фрагментов текста по сложным шаблонам.

APT (Annotation Processing Tool)

Дополнение к компилятору Java. Обеспечивает обработку аннотаций на основе модульной фабричной архитектуры; позволяет реализовать нестандартные аннотации, обрабатываемые во время компиляции.

AWT (Abstract Window Toolkit)

Традиционный платформенно-независимый инструментарий Java для работы с окнами, графикой и пользовательскими интерфейсами.

Boojum

Мистическое призрачное «альтер эго» Снарка. Из поэмы Льюиса Кэрролла «Охота на Снарка» (1876).

boolean

Примитивный тип данных Java с допустимыми значениями `true` и `false`.

byte

Примитивный тип данных Java; 8-разрядное число со знаком в дополнительном коде.

catch

Команда Java, открывающая блок обработки исключений после команды `try`. За ключевым словом `catch` следует один или несколько типов исключений и аргументов в круглых скобках, а также блок кода в фигурных скобках.

char

Примитивный тип данных Java; переменная типа `char` содержит один 16-разрядный символ Unicode.

classpath

Список путей, определяющих каталоги и файлы архивов со скомпилированными файлами классов Java и ресурсами для приложений. Эти пути просматриваются в процессе поиска компонентов приложений Java.

DOM (Document Object Model)

Представление в памяти полностью разобранного на элементы документа XML с помощью объектов `Element`, `Attribute`, `Text` и т. д. API XML DOM в Java стандартизируется консорциумом W3C (World Wide Web Consortium).

double

Примитивный тип данных Java; значение `double` представляет собой 64-разрядное (с двойной точностью) число с плавающей точкой в двоичном формате IEEE-754 (binary64).

DTD (Document Type Definition)

Документ на специализированном языке, регламентирующий структуру и атрибуты тегов XML. Спецификации DTD используются для проверки документов XML. Они могут ограничивать порядок и вложенность тегов, а также допустимые значения атрибутов.

EJB (Enterprise JavaBeans)

Архитектура серверных бизнес-компонентов (классов Java), названная по аналогии с архитектурой JavaBeans, но не имеющая к ней существенного отношения. В состав EJB входят бизнес-службы и компоненты баз данных, а также средства декларативной безопасности и поддержки транзакций.

enum

Ключевое слово Java для объявления класса-перечисления, содержащего список констант (их называют перечисляемыми константами). Обычно они используются в качестве идентификаторов и меток — как удобная и безопасная по отношению к типам альтернатива обычным числовым константам.

extends

Ключевое слово, используемое в объявлении класса для указания его суперкласса. Объявляемый класс получает доступ ко всем открытым (`public`)

и защищенным (`protected`) переменным и методам своего суперкласса; или, если объявляемый класс находится в том же пакете, он получает доступ ко всем переменным и методам суперкласса, кроме скрытых (`private`). Если в определении класса нет секции `extends`, то его суперклассом считается `java.lang.Object`.

final

Модификатор, который может применяться к классам, методам и переменным. В каждом из этих случаев он имеет похожий, но не идентичный смысл. Класс с модификатором `final` не может субклассироваться (расширяться). Например, `java.lang.System` является `final`-классом. Метод с модификатором `final` не может переопределяться в субклассе. Когда `final` применяется к переменной, она становится константой, то есть не может изменяться. (Если `final`-переменная указывает на объект, то это всегда будет один и тот же объект, хотя его содержимое может изменяться).

finalize

Зарезервированное имя метода. Метод `finalize()` вызывается виртуальной машиной Java, когда объект перестает использоваться (то есть когда на него не осталось ни одной ссылки), но до фактического освобождения занимаемой объектом памяти. В последнее время этот метод считается морально устаревшим из-за появления новых подходов (таких, как интерфейс `Closeable` и синтаксис «`try` с ресурсами»).

finally

Ключевое слово для определения блока `finally` в конструкции `try / catch / finally`. Блоки `catch` и `finally` обеспечивают обработку исключений и завершающие операции для кода в блоке `try`. Необязательный блок `finally` следует после блоков `try` и `catch`. Код блока `finally` выполняется один раз, независимо от того, как выполняется код блока `try`. При нормальном выполнении управление достигает конца блока `try` и переходит в блок `finally`, который обычно содержит все необходимые завершающие действия. Также см. *try* и *catch*.

float

Примитивный тип данных Java; значение `float` представляет собой 32-рядное (с одинарной точностью) число с плавающей точкой в формате IEEE-754.

HTTP (Hypertext Transfer Protocol)

Протокол, используемый веб-браузерами и другими клиентами для взаимодействия с веб-серверами. В простейшей форме этого протокола для запроса

файлов от сервера используются команды GET, а для отправки данных на сервер — команды POST.

IDE (Integrated Development Environment)

Интегрированная среда разработки программного обеспечения. Программа с графическим интерфейсом (например, IntelliJ IDEA или Eclipse), обеспечивающая написание и редактирование исходного кода, компиляцию, запуск, отладку и развертывание приложений Java.

implements

Ключевое слово, которое в объявлении класса означает, что этот класс реализует указанный интерфейс или интерфейсы. Секция `implements` не обязательна в объявлениях классов; если она присутствует, то должна следовать после секции `extends` (которая тоже необязательна). Если секция `implements` присутствует в объявлении неабстрактного класса, то каждый метод из каждого указанного интерфейса должен быть реализован в этом классе или в одном из его суперклассов.

import

Оператор `import` открывает доступ к сторонним классам Java под их сокращенными именами. Этот оператор необязателен, но удобен, так как избавляет от ввода лишних символов и упрощает чтение кода. Без помощи оператора `import` можно обращаться к классам Java по их полным именам, при условии, что соответствующие файлы классов могут быть найдены компилятором в `CLASSPATH` и доступны для чтения. Исходный код Java-программы может содержать любое количество операторов `import`. Все они должны быть расположены в начале файла, после необязательной команды `package` и до первого определения класса или интерфейса в файле.

import static

Оператор импортирования имен статических методов и переменных класса в область видимости класса, сходный с обычным оператором импортирования классов и пакетов. Статическое импортирование — вспомогательная конструкция, реализующая эффект глобальных методов и констант.

instanceof

Оператор Java, который возвращает `true`, если объект, указанный слева от него, является экземпляром класса (или реализует интерфейс класса), указанного в правой части. Оператор `instanceof` возвращает `false`, если объект не является экземпляром этого класса или не реализует этот интерфейс, а также если объект не определен (`null`).

int

Примитивный тип данных Java; 32-разрядное число со знаком в дополнительном коде.

interface

Ключевое слово, используемое для объявления интерфейса. Также см. *интерфейс*.

ISO 8859-1

8-разрядная кодировка символов, стандартизированная ISO (международной организацией по стандартизации). Эта кодировка также известна под названием Latin-1. Она содержит все символы латинского алфавита, которые есть в английском языке и в большинстве западноевропейских языков.

JavaBeans

Архитектура компонентов Java; предоставляет средства для создания взаимодействующих объектов, с которыми удобно работать в интегрированной среде разработки приложений.

JavaScript

Язык для создания динамических веб-страниц. Разработан компанией Netscape на заре истории интернета. С точки зрения программиста он не связан с Java, но некоторые синтаксические конструкции похожи.

JAXB (Java API for XML Binding)

Java API, позволяющий генерировать классы Java на основе описаний в форматах XML (DTD и Schema), а также генерировать XML-документы на основе классов Java.

JAXP (Java API for XML Processing)

Java API, позволяющий создавать подключаемые модули для обработки документов в форматах XML и XSL. Предоставляет механизм создания парсеров (анализаторов) и конвертеров, не зависящий от реализации.

JAX-RPC

Java API для работы с протоколом XML RPC (eXtensible Markup Language Remote Procedure Call), используемым веб-службами.

JDBC (Java Database Connectivity)

Стандартный Java API для работы с базами данных SQL (Structured Query Language).

JDK (Java Development Kit)

Пакет программ, распространяемый корпорацией Oracle для Java-разработчиков. Включает интерпретатор Java, классы Java и средства разработчика Java: компилятор, отладчик, дизассемблер, программу просмотра апплетов, генератор заглушек и генератор документации. Другое название — Java SDK (Software Development Kit).

JDOM

Пакет Java для работы с XML DOM (Document Object Model, объектная модель документа), созданный Джейсоном Хантером (Jason Hunter) и Бреттом Маклафлином (Brett McLaughlin). С пакетом JDOM проще работать, чем со стандартным DOM API для Java. JDOM использует API коллекций Java и стандартные соглашения Java.

JWS DP (Java Web Services Development Pack)

Традиционный набор стандартных API для разработки веб-служб. В JWS DP вошли пакеты JAXB, JAX-RPC, а также другие пакеты, относящиеся к XML и веб-службам.

Latin-1

Неофициальное название кодировки ISO 8859-1.

long

Примитивный тип данных Java; 64-разрядное число со знаком в дополнительном коде.

MIME-типы

Описанная в стандарте MIME система классификации типов данных, которые передаются в качестве вложений в сообщениях электронной почты и загружаются с серверов в качестве контента веб-страниц.

MVC (Model-View-Controller)

«Модель — представление — контроллер» — это архитектурный паттерн для пользовательских интерфейсов, впервые появившийся в Smalltalk. В MVC данные отображаемых на экране объектов называются моделью. Представлением называется конкретный способ визуализации этих данных, а контроллер обеспечивает взаимодействие пользователя с моделью и представлением. Концепции MVC часто встречаются в Java.

NaN (not-a-number)

«Не число». Специальное значение типов данных `double` и `float`, означающее неопределенный результат математической операции (например, деления 0 на 0).

new

Унарный оператор, создающий новый объект или массив (или выдающий исключение `OutOfMemoryException`, если для этой операции недостаточно памяти).

NIO

«Новый» (по сравнению с Java 1.0) пакет ввода-вывода Java, появившийся в Java 1.4. Фундаментальный пакет для поддержки асинхронных, прерываемых и масштабируемых операций ввода-вывода.

null

Специальное значение, означающее, что переменная ссылочного типа не указывает ни на какой экземпляр объекта. Статические переменные и переменные экземпляров по умолчанию содержат `null`, пока им не будет присвоено другое значение.

package

Оператор `package` указывает на пакет Java, содержащий классы Java. Код Java, который является частью пакета, имеет доступ ко всем классам этого пакета, а также ко всем методам и полям этих классов, кроме скрытых (помеченных модификатором `private`). Когда класс Java является частью именованного пакета, скомпилированный файл этого класса должен находиться в соответствующей позиции иерархии каталогов `CLASSPATH` до того, как к нему обратится интерпретатор Java или какая-либо другая программа. Если в файле с исходным кодом нет оператора `package`, то код этого файла считается частью безымянного пакета по умолчанию. Это удобно для небольших тестовых программ, запускаемых из командной строки. Это удобно и в процессе разработки, так как в этом случае весь код находится в каталоге по умолчанию.

printf

Стиль форматирования текста, происходящий из языка C. Основан на синтаксисе встроенных идентификаторов и на списках аргументов переменной длины для передачи параметров.

private

Ключевое слово `private` — модификатор видимости, применяемый к методам и полям классов. Помеченные этим модификатором методы и поля называются скрытыми, или приватными. Они не видны за пределами определений своих классов, и прямые обращения к ним из субклассов невозможны.

protected

Ключевое слово `protected` — модификатор видимости, применяемый к методам и полям классов. Помеченные этим модификатором поля называются защищенными. Они видны только внутри своего класса и его subclasses, а также в пределах пакета, частью которого является этот класс. Subclasses разных пакетов могут обращаться только к защищенным полям экземпляров своих классов или subclasses; они не могут обращаться к защищенным полям экземпляров суперкласса.

public

Ключевое слово `public` — модификатор видимости, применяемый к классам и интерфейсам, а также к методам и полям классов и интерфейсов. Помеченные этим модификатором классы и интерфейсы называются открытыми, они видны везде. Для сравнения, другие классы и интерфейсы видны только в пределах своего пакета. Метод или переменная с модификатором `public` видны везде, где виден их класс. Поле, не имеющее ни одного из модификаторов `private`, `protected` или `public`, видно только в пределах пакета, в который входит его класс.

SecurityManager

Класс Java, определяющий методы, которые система вызывает для проверки разрешений на те или иные операции в данной среде.

short

Примитивный тип данных Java: 16-разрядное число со знаком в дополнительном коде.

signature (сигнатура)

Идентификатор метода Java. Содержит имя метода и типы его аргументов, а также, возможно, возвращаемое значение. Совокупность этих атрибутов однозначно идентифицирует метод в некотором контексте.

static

Ключевое слово `static` — модификатор, применяемый в объявлениях методов и переменных в классе. Помеченные этим модификатором переменные называются статическими или переменными класса, в отличие от переменных экземпляра, которые не являются статическими. Хотя каждый экземпляр класса содержит свои собственные копии переменных экземпляра, существует только одна копия каждой статической переменной класса, независимо от количества созданных экземпляров этого класса (и даже при их отсутствии). К статической переменной можно обращаться через имя класса или через

экземпляр. А если переменная не является статической, то к ней можно обращаться только через экземпляр.

String

Класс Java, представляющий символьные данные (например, текст). В классе `String` есть много методов для работы с текстом и другими строками.

super

Ключевое слово, используемое классом для обращения к переменным и методам своего родительского класса. Ключевое слово `super` используется наподобие того, как ключевое слово `this` указывает на контекст «этого конкретного объекта».

synchronized

Ключевое слово, используемое в Java двумя взаимосвязанными способами: как модификатор и как оператор. Модификатор `synchronized` применяется к статическим методам и к методам экземпляров; он указывает, что метод изменяет внутреннее состояние класса или объекта таким способом, который не является потокобезопасным. Перед вызовом статического метода с модификатором `synchronized` Java захватывает блокировку соответствующего класса, чтобы другие потоки не могли одновременно изменять этот класс. Перед запуском метода экземпляра с модификатором `synchronized` Java захватывает блокировку объекта, вызвавшего метод, чтобы другие потоки не могли одновременно изменять этот объект. Модификатор `synchronized` гарантирует, что все изменения значений синхронизируются между потоками и в итоге становятся видимыми для всех ядер процессора. В Java также есть оператор `synchronized`, который указывает на «критическую секцию» в коде. За ключевым словом `synchronized` следует выражение в круглых скобках и одна или несколько команд. Результатом вычисления выражения должен быть объект или массив. Перед выполнением команд Java захватывает блокировку этого объекта или массива.

TCP (Transmission Control Protocol)

Надежный протокол, ориентированный на соединение. Один из основных протоколов интернета.

this

Ключевое слово, обозначающее в методе экземпляра или в конструкторе класса «этот конкретный объект», то есть тот экземпляр, с которым в настоящее время работает программа. Часто используется для ссылки на переменную

экземпляра, которая была замещена локальной переменной или аргументом метода. Также используется для передачи этого конкретного объекта в аргументе статическим методам или методам других классов. Кроме того, в первой команде конструктора ключевое слово `this` служит для указания на какой-либо другой конструктор класса.

throw

Команда `throw` «выбрасывает» заданный объект класса `Throwable` (объект исключения), тем самым сигнализируя, что в программе возникла аномальная ситуация. Эта команда останавливает обычное выполнение программы и передает управление ближайшей внешней команде `catch`, способной обработать заданный объект исключения. Также см. *catch*.

throws

Ключевое слово `throws` используется в объявлении метода для перечисления всех исключений, которые могут выдаваться этим методом. Любые исключения, которые могут выдаваться методом, но не являются subclasses `Error` или `RuntimeException`, должны либо перехватываться этим методом, либо объявляться в секции `throws` этого метода.

try

Ключевое слово `try` обозначает защищенный блок кода, к которому применяются последующие секции `catch` и `finally`. Сама команда `try` никаких специальных действий не выполняет. Также см. *catch* и *finally*.

try с ресурсами (try-with-resources)

Блок `try`, который открывает ресурсы, реализующие интерфейс `Closeable`, для их последующего автоматического освобождения. Также см. *try*.

UDP (User Datagram Protocol)

Ненадежный протокол, не ориентированный на соединение. UDP регламентирует соединение для передачи данных на основе дейтаграмм с минимальным контролем пакетов.

Unicode («Юникод»)

Универсальный стандарт кодирования текстовых символов, включающий в себя символы почти всех письменных человеческих языков. Стандартизацией занимается консорциум Unicode. Java использует Unicode для хранения данных двух типов: `char` и `String`.

UTF-8 (UCS Transformation Format, 8 bit)

Кодировка символов Unicode (в более общем виде — символов UCS), часто используемая для передачи и хранения данных. Это многобайтовый формат, в котором разные символы кодируются разным количеством байтов.

varargs

См. *список аргументов переменной длины*.

WAR (Web Applications Resources)

Формат файла, представляющего собой обычный JAR-архив с дополнительной структурой для хранения классов и ресурсов веб-приложений. Файл WAR содержит каталог `WEB-INF`, в котором хранятся классы, библиотеки и файл развертывания `web.xml`.

XInclude

Стандарт XML и Java API для включения дополнительных элементов в документы XML.

XML (Extensible Markup Language)

Универсальный язык разметки для текста и данных, использующий вложенные теги для структурирования контента и добавления метаданных.

XML Schema

Язык на основе XML, разработанный консорциумом W3C (World Wide Web Consortium). Позволяет регламентировать структуру XML-тегов и атрибутов тегов, а также структуру и типы данных, содержащихся в XML-документах. Есть и другие языки аналогичного назначения, их синтаксис отличается. Описание XML-документов с помощью XML Schema вытесняет старый стандарт DTD (Document Type Definition).

XPath

Стандарт XML и Java API для сопоставления элементов и атрибутов в XML-документах с помощью иерархического языка выражений, похожих на регулярные выражения.

XSL / XSLT (Extensible Stylesheet Language / XSL Transformations)

Основанный на XML язык, предназначенный для описания стилового оформления и преобразования XML-документов. В стиливое оформление входит простое добавление разметки, обычно для представления данных. XSLT позволяет в дополнение к стиловому оформлению полностью реструктурировать документы.

аннотации (annotations)

Метаданные с тегом @, добавляемые в исходный код Java. Аннотации могут использоваться компилятором, а также исполнительной системой (во время выполнения приложений) для расширения возможностей классов, для получения инструкций, для включения дополнительных сервисных функций.

атомарный (atomic)

Термин, означающий неделимость операции; операция выполняется как единое целое, по принципу «либо все, либо ничего». В Java некоторые операции виртуальной машины и некоторые операции API многопоточности являются атомарными.

веб-приложение (web application)

Приложение, работающее на веб-сервере или на сервере приложений, обычно в сочетании с пользовательскими веб-браузерами в качестве клиентов.

веб-служба (web service)

Служба уровня приложения, работающая на сервере. Обращение к веб-службам осуществляется через стандартный механизм: XML служит форматом передачи данных, а HTTP — сетевым транспортным протоколом.

вектор (vector)

Динамический массив элементов.

верификатор (verifier)

Своего рода механизм доказательства теорем, который анализирует байт-код Java перед его запуском и подтверждает, что байт-код работает корректно, не нарушая модель безопасности Java. Верификатор байт-кода — это первая линия защиты в модели безопасности Java.

внутренний класс (inner class)

Определение класса, вложенное в другой класс или метод. Внутренний класс функционирует в пределах лексической области видимости другого класса.

воплощение типа (type instantiation)

В обобщениях Java — та точка в коде, где обобщенный тип заменяется реальным. Другие варианты названия — инстанцирование типа, реализация типа, вызов типа. Также см. *обобщения*.

глубокая копия (deep copy)

Копия объекта вместе со всеми объектами, на которые он ссылается. В глубокой копии дублируется весь «граф» объектов, а не только ссылки. Также см. *поверхностная копия*.

границы (bounds)

В обобщениях Java — требования к реальному типу, который будет подставлен вместо параметра-типа. Верхняя граница означает, что этот тип должен расширять указанный класс Java (или быть совместимым с ним по присваиванию). Нижняя граница означает, что этот тип должен быть супертипом указанного типа (или быть совместимым с ним по присваиванию). Также см. *обобщения*.

графический контекст (graphics context)

Объект класса `java.awt.Graphics`, представляющий «поверхность для рисования». Содержит контекстную информацию о той области окна, которая определена для графического вывода, и предоставляет методы для рисования, то есть для графических операций.

графический пользовательский интерфейс (GUI)

Традиционный визуальный интерфейс пользователя, состоящий из окна с различными графическими элементами: кнопками, текстовыми полями, раскрывающимися меню, диалоговыми окнами и другими стандартными интерфейсными компонентами.

дайджест сообщения (message digest)

Значение, вычисленное из содержимого сообщения криптографическими средствами. Другие названия: профиль сообщения, хеш-код сообщения. Используется для определения того, было ли сообщение каким-то образом изменено. Любое изменение в содержимом сообщения приводит к изменению дайджеста сообщения. При правильной реализации практически невозможно создать два похожих сообщения с одинаковыми дайджестами.

дейтаграмма (datagram)

Пакет данных, обычно передаваемый по протоколу, не ориентированному на соединение (например, UDP). Такие протоколы не гарантируют доставку пакетов, не выявляют ошибки в процессе передачи и не передают управляющую информацию.

единица компиляции (compilation unit)

Единица исходного кода для класса Java. Единица компиляции обычно содержит определение одного класса и в большинстве современных сред разработки соответствует файлу с расширением `.java`.

журнальный API (Logging API)

Java API для обработки и структурированного вывода в журнал сообщений от компонентов приложения. Журнальный API поддерживает несколько уровней регистрации, соответствующих разным степеням важности сообщений, а также средства фильтрации.

загрузчик классов (class loader)

Экземпляр класса `java.lang.ClassLoader`, отвечающий за загрузку двоичных классов Java в виртуальную машину Java. Загрузчики классов помогают дифференцировать классы в целях структурирования и безопасности, а также могут объединяться в цепочки в иерархиях «родитель — потомок».

замещение (shadow)

Объявление переменной с таким же именем, как у переменной, определенной в суперклассе. Говорят, что такая переменная замещает (замещает собой) переменную суперкласса. Для ссылки на замещенную переменную используется ключевое слово `super` или преобразование объекта к типу суперкласса.

имя хоста (hostname)

Удобочитаемое имя (например, `oracle.com`), назначаемое компьютеру-серверу, подключенному к интернету.

инкапсуляция (encapsulation)

Техника объектно-ориентированного программирования, позволяющая ограничить доступ к переменным и методам для упрощения API класса или пакета. Используя ключевые слова `private` и `protected`, программист может ограничить видимость внутренних частей класса, чтобы класс работал по принципу «черного ящика». Инкапсуляция уменьшает риск ошибок, способствует повторному использованию классов и модульной структуре приложений. Эта техника также называется сокрытием данных (*data hiding*).

инстанцирование типа (type instantiation)

См. *воплощение типа*.

интернационализация (internationalization)

Процесс обеспечения доступности приложений для людей, говорящих на разных языках. Иногда обозначается сокращением «i18n» (в слове «internationalization» 18 букв между «i» и «n»).

интерпретатор (interpreter)

Модуль исполнительной системы Java, декодирующий и выполняющий байт-код Java. В современных версиях Java большая часть байт-кода Java не интерпретируется, а динамически компилируется в платформенный код виртуальной машины Java.

интерфейс (interface)

Набор абстрактных методов, которые в совокупности определяют ссылочный тип данных в языке Java. Класс, реализующий все эти методы, может объявить, что он реализует тип интерфейса, то есть его экземпляры являются объектами соответствующего типа. Также см. *interface*.

интроспекция (introspection)

В архитектуре JavaBeans — процесс, посредством которого какой-либо компонент предоставляет информацию о себе в дополнение к информации, которая может быть получена посредством отражения.

исключение (exception)

Сигнал о том, что в программе возникло непредвиденное состояние. В Java исключения — это объекты, которые являются subclasses *Exception* или *Error* (эти два класса — subclasses *Throwable*). Исключения в Java выдаются («выбрасываются») с помощью ключевого слова *throw* и обрабатываются с помощью ключевого слова *catch*. Также см. *catch*, *throw* и *throws*.

класс (class)

1. Фундаментальная единица, определяющая объект в большинстве объектно-ориентированных языков программирования. Класс представляет собой инкапсулированный набор переменных и методов, которые могут иметь привилегированный доступ друг к другу. Обычно на основе класса создаются экземпляры (объекты), содержащие собственные наборы данных.
2. Ключевое слово *class* используется, чтобы объявить класс и тем самым определить новый тип объектов.

клиент (client)

Потребитель ресурса или сторона, инициирующая взаимодействие в случае сетевого клиент-серверного приложения. Также см. *сервер*.

компилятор (compiler)

Программа, преобразующая исходный код в исполняемый.

композиция (composition)

Объединение существующих объектов для создания другого, более сложного объекта. Создавая новый объект посредством композиции, программист формирует сложное поведение, делегируя задачи внутренним объектам. Композиция отличается от наследования, при котором новый объект определяется посредством изменения или уточнения поведения существующего объекта. Также см. *наследование*.

компонентная архитектура (component architecture)

Методология создания отдельных частей приложения; способ создания пригодных для повторного использования объектов, из которых легко составлять типовые приложения.

компоненты JavaBeans

Классы Java, написанные в соответствии с паттернами проектирования и соглашениями пакета JavaBeans.

конструктор (constructor)

Специальный метод, который вызывается автоматически при создании нового экземпляра класса. Конструкторы используются для инициализации переменных только что созданного объекта. Имя конструктора всегда совпадает с именем класса, и конструктор не имеет явного возвращаемого значения.

контекст сервлетов (servlet context)

В API сервлетов — среда веб-приложения, предоставляющая сервлету ресурсы сервера и приложения. Кроме того, контекстом сервлета часто называют базовый URL-адрес веб-приложения.

криптография с открытым ключом (public-key cryptography)

Криптографическая система, для работы которой необходимы два ключа: открытый и закрытый. Закрытый ключ позволяет расшифровать сообщения, зашифрованные с соответствующим открытым ключом, и наоборот. Открытый ключ может публиковаться без риска для конфиденциальности и использоваться для проверки подлинности сообщений, отправленных с соответствующим закрытым ключом.

локальная переменная (local variable)

Переменная, объявляемая внутри метода. Локальная переменная видна только в коде этого метода.

лямбда-выражение (lambda expression)

Компактная запись для размещения всего определения небольшой анонимной функции непосредственно в том месте кода, где она используется.

менеджер макета (layout manager)

Объект, управляющий размещением компонентов внутри отображаемой области контейнера Swing или AWT.

метод (method)

Название функции или процедуры в объектно-ориентированном программировании.

метод класса (class method)

См. *статический метод*.

метод экземпляра (instance method)

Метод класса, не имеющий модификатора `static`. Такому методу неявно передается ссылка на текущий объект, для которого он был вызван. Также см. *static*, *статический метод*.

модификатор (modifier)

Ключевое слово, размещаемое в исходном коде перед классом, переменной или методом и изменяющее уровень доступности, поведение или смысл этого элемента. Также см. *abstract*, *final*, *private*, *protected*, *public*, *static*, *synchronized*.

наследование (inheritance)

Важный механизм объектно-ориентированного программирования, при котором новый объект определяется изменением или уточнением поведения существующего. При наследовании в новый объект неявно включаются все переменные и методы его родительского объекта, кроме скрытых (имеющих модификатор `private`). В данном контексте под объектами имеются в виду классы и интерфейсы. Java поддерживает одиночное наследование классов и множественное наследование интерфейсов.

необработанный тип (raw type)

В обобщениях Java — простой тип класса Java без какой-либо информации об обобщенном типе. Необработанные типы — это реальные типы всех классов Java после компиляции. Также см. *стирание типа*.

облегченный компонент (lightweight component)

Написанный на Java компонент пользовательского интерфейса (GUI-компонент), не имеющий платформенного аналога в AWT.

обобщения (generics)

Синтаксис и реализация параметризованных (обобщенных) типов в языке Java, начиная с Java 5.0. Обобщенные типы — это классы Java, поведение которых можно специализировать, передавая им в параметрах имена одного или нескольких ссылочных типов Java. Обобщения похожи на шаблоны (templates), существующие в некоторых других языках. Также см. *обобщенный класс*, *обобщенный метод*.

обобщенный класс (generic class)

Класс, содержащий в исходном коде одно или несколько условных обозначений, которые называются «параметры-типы» (type parameters) или «переменные-типы» (type variables). При использовании класса компилятор подставляет вместо них те имена ссылочных типов, которые указал программист. Обобщенные классы особенно полезны для объектов-контейнеров и коллекций, легко адаптируемых для работы с элементами нужных типов.

обобщенный метод (generic method)

Метод, содержащий в исходном коде одно или несколько условных обозначений, которые называются «параметры-типы» (type parameters) или «переменные-типы» (type variables). При использовании метода компилятор подставляет вместо них те имена ссылочных типов, которые указал программист. Во многих случаях компилятор автоматически определяет нужные типы по контексту использования метода.

обработчик контента (content handler)

Класс, который вызывается для разбора (парсинга, анализа) данных определенного типа и для их преобразования в соответствующий объект.

обработчик протокола (protocol handler)

Компонент URL, реализующий сетевое соединение, необходимое для обращения к ресурсу по URL-адресу (например, HTTP или FTP). Обработчик протокола Java состоит из двух классов: `StreamHandler` и `URLConnection`.

обратный вызов (callback)

Поведение, которое определяется в одном объекте, а затем активизируется другим объектом при возникновении конкретного события. Механизм событий Java является разновидностью схемы обратного вызова.

объект (object)

1. Фундаментальная структурная единица объектно-ориентированного языка программирования, инкапсулирующая набор данных и логику работы с ними.
2. Экземпляр класса, который имеет структуру класса, но содержит собственные копии элементов данных. Также см. *экземпляр*.

отражение (reflection)

Способность языка программирования взаимодействовать со своими собственными структурами во время выполнения программы. Механизм отражения позволяет исполнительной системе Java анализировать файлы классов во время выполнения, получать информацию об их методах и переменных, динамически вызывать методы и изменять переменные.

очередь (queue)

Структура данных, напоминающая список; часто используется по принципу FIFO (First In, First Out — «первым пришел, первым ушел») для буферизации обрабатываемых элементов.

параметризованный тип (parameterized type)

Класс, использующий синтаксис обобщений Java и компилируемый в зависимости от одного или нескольких ссылочных типов, указанных программистом. Их имена автоматически подставляются в объявление класса и в тело класса вместо соответствующих обозначений («параметров-типов» или «переменных-типов»), и таким образом класс адаптируется к работе с этими типами. Также см. *универсальный тип*.

перегрузка методов (method overloading)

Механизм определения нескольких методов с одним и тем же именем, но с разными списками аргументов. При вызове перегруженного метода компилятор выбирает нужную версию на основании типов переданных аргументов.

переменная класса (class variable)

См. *статическая переменная*.

переменная экземпляра (instance variable)

Переменная класса, не имеющая модификатора *static*. Независимые копии всех таких переменных есть в каждом экземпляре класса. Также см. *переменная класса*, *static*.

переопределение метода (method overriding)

Определение метода с таким же именем и с такими же типами аргументов, как у метода, определенного в суперклассе. При вызове переопределенного метода интерпретатор использует динамический поиск для определения того, какое определение метода применимо к текущему объекту. Начиная с Java 5.0, переопределенные методы могут иметь разные возвращаемые типы, с некоторыми ограничениями.

перечисление (enumeration)

См. *enum*.

платформенный метод (native method)

Метод, реализованный на низкоуровневом языке платформы (операционной системы), а не на языке Java. Платформенные методы предоставляют доступ к таким ресурсам, как сеть, оконная система и файловая система компьютера.

поверхностная копия (shallow copy)

Копия объекта, дублирующая только значения, содержащиеся в самом объекте. В ней, в частности, дублируются ссылки на другие объекты, но при этом копирования самих этих объектов не происходит. Также см. *глубокая копия*.

подписанный апплет (signed applet)

Апплет, упакованный в JAR-файл и снабженный цифровой подписью, позволяющей проверить подлинность его происхождения и целостность содержимого.

подписанный класс (signed class)

Класс Java (в том числе в составе архива Java), дополненный цифровой подписью. Подпись позволяет получателю проверить происхождение класса и убедиться в том, что он не был изменен. Такому классу получатель может предоставить более высокие привилегии во время выполнения программы.

полиморфизм

Один из фундаментальных принципов объектно-ориентированного языка. Он заключается в том, что тип, который расширяет другой тип, является «частным случаем» этого родительского типа и может использоваться в качестве его замены, расширяя или уточняя его функциональность.

поток (thread)

Независимая ветвь выполнения кода в программе. Java является многопоточным языком программирования, то есть в интерпретаторе Java могут выполняться сразу несколько потоков. Для представления потоков и для управления ими в Java используются объекты Thread.

поток данных (stream)

Последовательность данных, передаваемых по каналу связи. Все фундаментальные операции ввода-вывода в Java основаны на потоках данных. Классы NIO используют каналы, ориентированные на работу с пакетами данных. В Java 8 появился API потоков данных (Stream API), ориентированный на функциональный стиль программирования.

преобразование типа (cast)

Изменение видимого типа объекта Java, приведение к какому-либо другому указанному типу. Преобразования типов Java проверяются как статически (компилятором Java), так и во время выполнения приложений.

приложение (application)

Java-программа, которая выполняется автономно (в отличие от апплета).

примитивный тип (primitive type)

Один из следующих типов данных Java: boolean, char, byte, short, int, long, float и double. Данные примитивных типов обрабатываются, присваиваются и передаются методом «по значению», то есть при этом копируются байты, содержащие эти данные. Также см. *ссылочный тип*.

проверочные утверждения (assertions)

Средства языка, используемые для проверки условий, истинность которых должна обеспечиваться логикой программы. Если условие, проверяемое проверочным утверждением, окажется ложным, то возникает фатальная ошибка. При разворачивании приложения проверочные утверждения можно отключить для повышения быстродействия.

пул потоков (thread pool)

Группа потоков, рассчитанных на повторное использование для обслуживания рабочих запросов. Поток назначается для обработки какого-то одного элемента, после чего возвращается в пул.

распаковка (unboxing)

Извлечение из объектов-обертки инкапсулированных в них значений примитивных типов данных (например, `int`).

регулярные выражения (regular expressions)

Компактный, но мощный синтаксис описания шаблонов для текста. Регулярные выражения могут использоваться для поиска и разбора (парсинга) множества разнообразных текстовых конструкций.

сервер (server)

Сторона, предоставляющая ресурсы или получающая запросы на взаимодействие в клиент-серверных приложениях. Также см. *клиент*.

сервлет (servlet)

Компонент приложения Java, реализующий API сервлетов (`javax.servlet.Servlet`). Может выполняться в контейнере сервлетов или на веб-сервере. Сервлеты широко используются в веб-приложениях для обработки пользовательских данных и для создания HTML-документов и других данных, передаваемых веб-браузерам.

сериализовать (serialize)

Упорядочить, сделать последовательным. Сериализованные методы — это методы, синхронизированные с потоками таким образом, что только один из них может выполняться в каждый момент времени.

сертификат

Электронный документ, использующий цифровую подпись для идентификации человека или организации. Сертификат подтверждает действия от имени конкретного человека или группы людей и содержит открытый ключ. Сертификаты создают и выдают центры сертификации, подписывая их своими цифровыми подписями. Также см. *цифровая подпись*.

событие (event)

1. Действие пользователя, например щелчок кнопкой мыши или нажатие клавиши.
2. Объект Java, передаваемый зарегистрированному слушателю событий в ответ на действие пользователя или другую активность в системе.

сокеты (sockets)

Сетевой API, происходящий из BSD Unix. Два сокета становятся конечными точками для обмена данными между двумя сторонами сетевого соединения. Серверный сокет прослушивает запросы на соединения от клиентов и создает отдельный сокет на своей стороне для каждого соединения.

сокрытие данных (data hiding)

См. *инкапсуляция*.

спиннер (spinner)

Компонент графического интерфейса, который отображает значение и рядом с ним — две маленькие кнопки со стрелками вверх и вниз, которыми можно «перелистывать» это значение (как правило, увеличивать и уменьшать). Компонент `JSpinner` из пакета `Swing` может работать с диапазонами чисел, с датами и с произвольными перечислениями.

список аргументов переменной длины (variable-length argument list)

Метод Java может объявить, что он может получать после обязательных аргументов любое количество дополнительных аргументов указанного типа. Такие аргументы обрабатываются с помощью их упаковки в массив.

ссылочный тип (reference type)

Любой объект или массив. Данные ссылочных типов обрабатываются, присваиваются и передаются методам «по ссылке», то есть при этом копируются не сами эти данные, а ссылки на них. Также см. *примитивный тип*.

статическая переменная (static variable)

Переменная, объявленная с ключевым словом `static`. Она является переменной всего класса (а не какого-то конкретного экземпляра этого класса). Независимо от количества созданных экземпляров класса, существует только одна копия статической переменной.

статический метод (static method)

Метод, объявленный с ключевым словом `static`. Он является методом всего класса (а не какого-то конкретного экземпляра), к нему неприменима ссылка `this`, он может обращаться только к статическим переменным и вызывать другие статические методы класса. Статический метод вызывается через имя класса, а не через экземпляр класса.

стирание типа (type erasure)

Техника реализации, используемая в обобщениях Java; заключается в том, что при компиляции удаляется (стирается) информация об обобщенных типах и они преобразуются в необработанные типы (raw types) Java. Стирание типа обеспечивает обратную совместимость с «необобщенным» кодом Java, но создает некоторые затруднения в языке. Также см. *необработанный тип*.

субкласс (subclass)

Класс, расширяющий другой класс, который в этом случае называется суперклассом. Субкласс наследует открытые и защищенные методы и переменные своего суперкласса. Также см. *extends*.

суперкласс (superclass)

Родительский класс, расширяемый другим классом (субклассом). Открытые (public) и защищенные (protected) методы и переменные суперкласса доступны для субкласса. Также см. *extends*.

сцепление исключений (exception chaining)

Паттерн программирования, заключающийся в перехвате исключения и выдаче нового исключения более высокого уровня или более подходящего исключения, которое указывает на исходное исключение как причину своего возникновения. При необходимости программист может получить объект исходного исключения.

уборка мусора (garbage collection)

Процесс освобождения памяти от неиспользуемых объектов. Объект считается неиспользуемым, если на него не осталось ни одной ссылки в других объектах и в локальных переменных из стека вызова методов любого потока.

универсальный тип (wildcard type)

В обобщениях Java — синтаксический элемент ? (знак вопроса), которым обозначают, что параметр-тип воплощается (инстанцируется) не конкретным типом, а каким-либо из множества типов, заданных в этой же синтаксической конструкции.

упаковка (boxing)

Инкапсуляция значений примитивных типов данных Java (например, int) в объектах-обертках, соответствующих этим типам. Также см. *распаковка*.

хеш-код

Идентификатор конкретного объекта (внешне похожий на случайное число), вычисляемый по содержащимся в объекте данным; своего рода подпись объекта. Хеш-коды используются для хранения объектов в хеш-таблице. Также см. *хеш-таблица*.

хеш-таблица

Объект, напоминающий словарь или ассоциативный массив. Другое название — хеш-карта. Для сохранения и чтения элементов из хеш-таблицы используются ключи, называемые хеш-кодами.

центр сертификации

Организация, которой доверяется выдача сертификатов и выполнение действий, необходимых для идентификации людей и организаций.

цифровая подпись (digital signature)

Комбинация дайджеста сообщения, зашифрованного закрытым ключом подписывающей стороны, и сертификата подписывающей стороны, идентифицирующего личность или организацию. Получатель подписанного сообщения может получить из сертификата открытый ключ подписывающей стороны, расшифровать дайджест зашифрованного сообщения и сравнить результат с дайджестом, вычисленным по подписанному сообщению. Если два дайджеста совпадут, то получатель будет знать, что сообщение не было изменено, а подписывающая сторона является именно тем, за кого себя выдает. Также см. *центр сертификации*.

экземпляр (instance)

Как правило, имеется в виду экземпляр класса. Когда на основе какого-либо класса создается объект, этот объект называют экземпляром этого класса.

Об авторах

Марк Лой (Marc Loy) «подхватил вирус» Java в 1994 году после знакомства с бета-версией браузера HotJava, в которой крутилась анимация алгоритма сортировки. Разрабатывал и проводил обучающие курсы в Sun Microsystems, а впоследствии продолжал обучать намного более широкую аудиторию. Сейчас он занимается консультациями и написанием материалов по техническим и медийным темам. Также он исследует быстро растущий мир встраиваемой электроники и носимых устройств.

Патрик Нимайер (Patrick Niemeyer) начал участвовать в разработке языка Oak («предка» Java) во время работы в Southwestern Bell Technology Resources. В настоящее время является техническим директором Ikeyzo, Inc., а также независимым консультантом и автором. Патрик создал BeanShell — популярный язык сценариев для Java. Он был участником нескольких экспертных групп JCP, которые руководили проектированием функциональности языка Java, и внес свой вклад во многие проекты с открытым кодом. В последнее время Патрик занимается разработкой аналитических программ для финансовой отрасли, а также современных мобильных приложений. Живет в Сент-Луисе с семьей и несколькими питомцами.

Дэн Лук (Dan Leuck) — исполнительный директор Ikeyzo, Inc., компании интерактивного проектирования и разработки программного обеспечения с филиалами в Токио и Гонолулу (среди клиентов Ikeyzo — Sony, Oracle, Nomura, PIMCO и федеральное правительство). Ранее работал вице-президентом по научным исследованиям и разработке в компании ValueCommerce (Токио), крупнейшей азиатской компании онлайн-маркетинга; руководителем по разработке в LastMinute.com (Лондон), крупнейшего европейского сайта B2C; президентом филиала DML в США. Дэн работал во многих консультативных советах и в комитетах компаний Macromedia и Sun Microsystems. Он является активным участником сообщества Java, работает над BeanShell, руководит проектом SDL и входит во многие экспертные группы Java Community Process.

Иллюстрация на обложке

На обложке книги изображен бенгальский тигр с детенышами. Бенгальский тигр — подвид тигров (*Panthera tigris tigris*), обитающий в Южной Азии. В результате охоты бенгальские тигры были почти полностью истреблены, а оставшиеся живут в основном в природных заповедниках и в национальных парках под строгой охраной. Считается, что в дикой природе осталось менее 3500 бенгальских тигров.

Бенгальский тигр имеет красновато-оранжевый окрас с узкими черными, серыми и коричневыми полосами, большей частью вертикальными. Длина тела взрослого самца достигает 2,7 метра, а вес превышает 200 килограммов. Это самые крупные из живущих на Земле представителей семейства кошачьих. Предпочитаемая ими среда обитания — густые рощи, высокая трава или заросли тамариска вдоль речных берегов. Максимальная продолжительность жизни может достигать 26 лет, но в дикой природе обычно не превышает 15 лет.

Тигрята обычно рождаются между февралем и маем — после беременности продолжительностью 3,5 месяца. Самки дают один помёт каждые 2–3 года. При рождении тигрята весят чуть больше килограмма и уже имеют полосатый окрас. Один помёт включает от одного до четырех тигрят, но выживает обычно не более двух-трех. Вскармливание обычно прекращается через 4–6 месяцев, но мать продолжает обеспечивать тигрят едой и защитой еще два года. У самок период зрелости наступает в возрасте от 3 до 4 лет, у самцов — от 4 до 5 лет.

Бенгальские тигры находятся под угрозой исчезновения из-за браконьеров, а также из-за разрушения и фрагментации их природной среды обитания. Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой исчезновения; все они важны для нашей планеты.

Цветная иллюстрация нарисована Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры неизвестного происхождения.

Марк Лой, Патрик Нимайер, Дэниэл Лук

Программируем на Java

5-е международное издание

Перевел с английского Е. Матвеев

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>В. Здобнов</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, Н. Викторова</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортёр в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 19.08.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 43,860. Тираж 1000. Заказ 0000.

Дэвид Копец

КЛАССИЧЕСКИЕ ЗАДАЧИ COMPUTER SCIENCE НА ЯЗЫКЕ JAVA



Столкнулись с «неразрешимой» проблемой при разработке программного обеспечения? Скорее всего, кто-то уже справился с этой задачей и вы можете не ломать голову. Дэвид Копец собрал наиболее полезные готовые решения, принципы и алгоритмы. «Классические задачи Computer Science на языке Java» — это мастер-класс по программированию, содержащий 55 практических примеров, затрагивающих самые актуальные темы: базовые алгоритмы, ограничения, искусственный интеллект и многое другое.

КУПИТЬ

Мартин Одерски, Лекс Спун, Билл Веннерс, Фрэнк Коммерс

SCALA. ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

5-е издание



«Scala. Профессиональное программирование» — главная книга по Scala, популярному языку для платформы Java, в котором сочетаются концепции объектно-ориентированного и функционального программирования, благодаря чему он превращается в уникальное и мощное средство разработки.

Этот авторитетный труд, написанный создателями Scala, поможет вам пошагово изучить язык и идеи, лежащие в его основе.

Пятое издание значительно обновлено, чтобы охватить многочисленные изменения, появившиеся в Scala 3.

КУПИТЬ

Роберт Седжвик, Кевин Уэйн

COMPUTER SCIENCE: ОСНОВЫ ПРОГРАММИРОВАНИЯ НА JAVA, ООП, АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ



Преподаватели Принстонского университета Роберт Седжвик и Кевин Уэйн создали универсальное введение в Computer Science на языке Java, которое идеально подходит как студентам, так и профессионалам. Вы начнете с основ, освоите современный курс объектно-ориентированного программирования и перейдете к концепциям более высокого уровня: алгоритмам и структурам данных, теории вычислений и архитектуре компьютеров.

И главное — вся теория рассматривается на практических и ярких примерах: прикладная математика, физика и биология, числовые методы, визуализация данных, синтез звука, обработка графики, финансовое моделирование и многое другое.

КУПИТЬ